KLayout 0.29.0 Documentation

2024-03-31, Matthias Köfferlein

https://www.klayout.org

Table of Contents

Main Index

Welcome to KLayout's documentation

The documentation is organized in chapters. For a brief introduction read the User Manual. 'Various Topics' is a collection of brief articles about specific topics. For Ruby programming see the 'Programming Ruby Scripts' chapter and for a complete Ruby class reference see the 'Class Index'.

1. KLayout User Manual

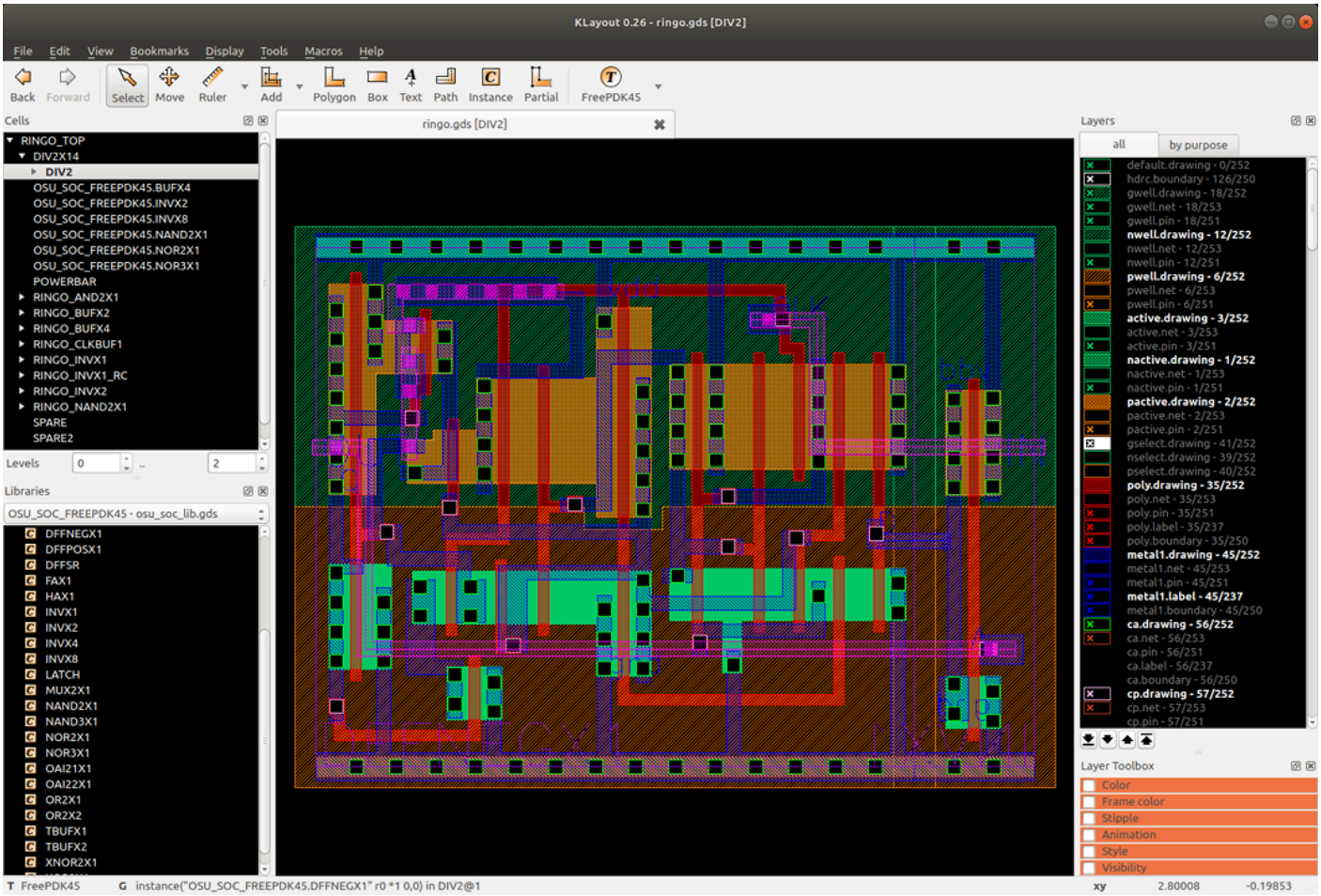
This is KLayout's main user manual. The manual is organised in major topics:

1.1. KLayout Basics

Welcome to KLayout's user manual. This is the manual chapter covering the basic features of KLayout. The following subtopics are available:

1.1.1. The Main Window

The main window is divided into three parts: the left area is the hierarchy browser and navigator, the center part of the canvas and the right part is the layer list with the layer toolbox. The individual components can be rearranged, so the arrangement described is just the default arrangement. You can move a component to a new place by dragging it with its title bar to some other place or detach it from the main window to form a floating separate window.

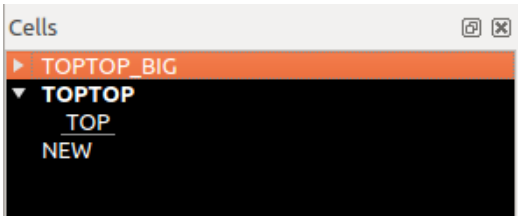


Left Part - The Hierarchy Browser, Library View and Navigator

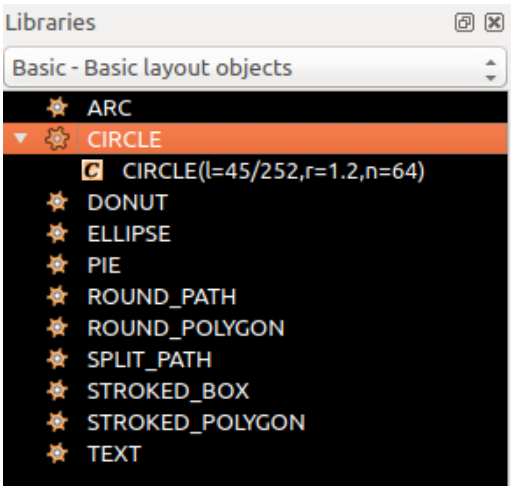
The left panel is the hierarchy browser which shows the cell hierarchy and - optionally - the navigator window that shows an overview over the whole layout.

In the hierarchy browser, cell nodes can be expanded showing the child nodes. The "current cell" is the one shown in the center panel. It is drawn in bold font. One or multiple cells can be selected. The selected cells are the ones, the various functions act on. The "context cell" is the cell which is the "active cell" on which drawing happens. The context usually is the same than the current cell, but by descending into the hierarchy, the child cell of the current cell can be made the context cell. It is shown in underlined font.

In the following example, "TOPTOP_BIG" is selected, "TOPTOP" is the current cell and "TOP" is the context cell:

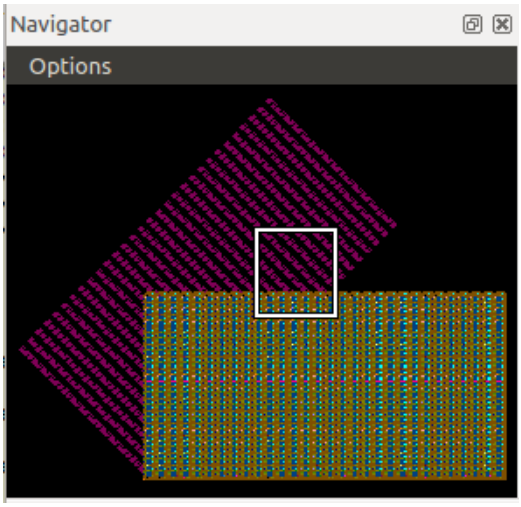


The sub-panel below the hierarchy browser is the library view. This view shows the libraries registered in the system and their content:



The library view is used to browse and place library cells, which can be normal cells or PCells. Read more about the library view here: The Library View.

The navigator is invisible by default but can be activated by checking the "Navigator" menu item in the "View" menu. The navigator shows an overview image of the whole cell and a box indicating the clip shown in the center panel:



Center Part - The Canvas

The center panel is the actual canvas. There, the layout is drawn. To zoom in, click with the right mouse button and drag a rectangle that will become the new area shown. Select items by left-clicking or dragging a selection rectangle with the left mouse button pressed. A variety of edit and display modification feature is supplied, for example the ability to add rulers or background images.

Right Part - The Layer List and Layer Style Controls

The right panel is shows the layer list or layer tree. By default, it shows a plain layer list, but the are manifold ways to configure the list, i.e. grouping, styling, adding tabs to easily switch between different setups etc.

The layer tree specifies what and how layout is drawn. Entries can be deleted or configured freely - this does not affect the layout itself. Essentially, the layer list can be entirely independent from the layout and omit layers from the layout or add (empty) layers that are not actually part of the layout. In this scenario, a layer list typically reflects a set of drawing layers with a certain technology-dependent meaning. This list provides a styled layer view for an otherwise undecorated layout file. The layers can be reordered, so their drawing priority is changed. The top layer will be drawn first, while others will be drawn later. Hence the first layer is likely to become obscured by following layers.

Modification operators can be attached to layers - for example, the drawn layout can be transformed geometrical (i.e. translated, rotated, scaled, mirrored) and layout can be selectively drawn when certain conditions for user properties apply.

Below the layer list, a set of control panels is located. The control panels are minimized per default. They can be expanded by clicking on the header bar. These controls form the "Layer Toolbox" where you can modify the layer styles. The styles selected in the layer toolbox will affect the layers selected in the layer list.

Multiple layouts can be shown together. Either they can be overlayed or they can be shown in separate views. In this case, a tab panel appears at top of the main window. Selecting a tab switches between the layouts. Layers for different layouts appear as annotated layers in the layer list - for example "1/0@1" for layer 1, datatype 0, first layout and "1/0@2" for layer 1, datatype 0 and second layout.

1.1.2. Loading A File

In the "File" menu, choose

- "Open" to close the current view and open a new layout instead of the currently loaded one
- "Open in Same Panel" to open a new layout in addition to the currently loaded one
- "Open in New Panel" to open a new layout in a new view

Either way, a file selection dialog will appear where a file can be chosen for loading. After choosing the file and clicking "Ok", the file is loaded.

The program will automatically determine the type of the file. Currently, OASIS, GDS2, DXF, CIF, LEF/DEF and a text version of GDS2 are supported. Gerber PCB data can be read with some preparations too (see Import Gerber PCB Files). If the file is gzip/zlib compressed, it will be uncompressed automatically.

Certain options can be specified for the file loader using the reader option pages. To open the reader options dialog, choose "Reader Options" from the "File" menu. This dialog allows specification of certain options for all "Open" actions, for example:

- Confine the reader to a certain set of layers. All other layers are not read.
- Disable reading of text objects. Text objects don't carry geometrical information for masks and can be discarded this way.
- Disable reading of user properties. If properties are not required, the memory consumption can be reduced by disabling properties.
- Certain GDS specific options which mainly control the level of compatibility with other tools.
- Other formats may offer other options too. Specifically rich formats such as DXF or LEF/DEF can be configured in manifold ways. Different tabs show options for different formats or format groups.

Using "File/Reload", the currently loaded file can be re-read from disk. Usually this is not required, because KLayout will automatically check whether the file has changed and offer to load it.

By picking a file from the "Open Recent" list in the "File" menu, a previous file can be loaded again.

Files can be given to KLayout on the command line and are loaded automatically. Multiple files can be specified. They are shown in different pages by default. To load multiple files into the same page, add a "-s" option to the command line. "http:" or "https:" can be specified on the command line as well. In this case, KLayout will download the files from the given URL.

Files and URL's can be dragged and dropped on the KLayout main window. KLayout will then load and show these files.

Layout files can be associated with technologies. Technologies allow associating a layout with additional data, such as libraries, macros, net tracer settings, layer properties etc. Read About Technology Management for details.

1.1.3. Managing The Panels And Loaded Layouts

Choose "Close" in the "File" menu to remove a layout of a panel and close the panel unless there are still layouts loaded. If multiple layouts are loaded into the current panel, a dialog appears. This allows selecting one or many layouts for closing. "Close All" will close all panels.

Choose "Clone" from the "File" menu to duplicate a panel. A new panel will be created that is an exact copy of the current one. Both, the current and the new panel are views to the same layout. This way, only one copy of the layout is held in memory.

Choose "Pull In Other Layout" to combine other layouts already loaded into the current panel. Basically, KLayout allows viewsing a layout in multiple panels, either on its own in different configurations or together with other layouts. "Pull In Other Layout" allows configuration of a panel to show another layout which has been loaded into another panel. In that sense it's the reverse of closing one layout from a panel showing multiple layouts.

1.1.4. Choosing A Cell

To show a certain cell, select the cell in the cell hierarchy in hierarchy browser to the left. Then, right-click in the cell tree to bring up the context menu and choose "Show as top" or simply select the cell with the middle mouse button.

To select a cell by name, choose "Select Cell" in the "Display" menu. A dialog will appear that allows selecting a cell by name or choosing it from an alphabetically sorted list. Additionally, this dialog allows navigating the cell tree by moving to one of the child or parent cells.

1.1.5. Choosing A Hierarchy Depth

By default, only the bounding box of the cell selected is shown. This corresponds to zero hierarchy levels being shown. If you select one more level of hierarchy (Levels 0 to 1), this content of the child cells of the current cell are drawn, but grandchildren (children of child cells) are drawn as boxes. Increasing the hierarchy levels will draw more and more details of child cells until all cells below the current cell are drawn in detail.

To select more hierarchy levels

- Select "Full Hierarchy" from the "Display" menu or press the "*" key to show all hierarchy levels
- Select "Box Only" from the "Display" menu or press the "0" key to show only the bounding box (the default)
- Select "Top Level Only" from the "Display" menu or press the "1" key to show the top level elements
- Select "Increment Hierarchy" from the "Display" menu or press the "+" key to show one more hierarchy level
- Select "Decrement Hierarchy" from the "Display" menu or press the "-" key to show one hierarchy level less
- Use the hierarchy level entry fields below the cell list to change the current minimum or maximum level

1.1.6. Configuring The Cell List

Two modes are provided for the cell list: a tree view (the default) and a flat cell list. To switch to flat mode, check the "Flat Cell List" option in the cell panel's context menu.

If multiple layouts are loaded, the cell lists of the individual layouts are shown separately. The drop-down box above the cell lists will select the current cell tree to show. An alternative mode is available in which the cell lists are shown beside each other in the cell tree panel. This mode is enabled by choosing the "Split Mode" option in the cell panel's context menu. In split mode, you can click on the headers to select the current cell tree. The current cell tree plays a role in some cases, for example for the layout operations available in the "Layout" submenu of the "Edit" menu.

In addition, three sorting modes are provided: alphabetically by name and by cell size (bounding box area), descending and ascending. The cell size is supposed to reflect the design level: library and leaf cells are usually small which macro blocks are usually large. By using cell size sorting in ascending order, the leaf cells will be shown first. To change the sorting, check the corresponding option in the "Sorting" submenu of the cell panel's context menu.

1.1.7. The Library View

Beside the cell list, a library view is provided as a support view for the layout panel. This sub-panel displays the libraries available and allows browsing the cells and PCells inside a library.

By default, the library view is shown below the cell tree. You can rearrange the views by dragging them at their title bar and docking them in other places of the main window. To reset the window arrangement to the default configuration, use "Restore Window" from the "View" menu.

The library view shows the cells and PCells of one library. To select a library, choose it from the selection box at the top of the library view.

PCells are shown with a small "gear" icon in the library view. If PCells are instantiated, the variants in use are shown as entries below the PCell entry.

In edit mode, cells can conveniently be placed by dragging them from the library view to the layout canvas. If a PCell variant is dragged, another instance of this PCell variant is created. If a PCell master is dragged, KLayout will pop up the PCell parameter definition dialog on drop.

1.1.8. Hiding Cells

Independent of the hierarchy levels shown, cells can be hidden. In this case, the cell is now shown but rather the bounding box is shown. To do so, select the cell from the cell list and choose "Hide" from the context menu. To show a cell again, choose "Show". To make all cells visible again, choose "Show All".

1.1.9. Zooming Into The Layout

Select the zoom area with the right mouse button in the layout canvas.

Press the button, drag the box to the desired position and release the button.

To zoom in (enlarge) drag the box right and down. To zoom out (shrink) drag the box up and left. To choose a new center, single-click the new center point with the right mouse button.

Additionally, these functions are available from the menu or by hotkeys:

- Pan to the left, right, top or bottom using the arrow keys or choosing one of these functions from the "Display" menu.
- Fill the whole selected cell into the window by pressing "F2" or choosing "Zoom Fit" from the "Display" menu
- Zoom in or out by a fixed amount by pressing "Enter" or "Shift+Enter" or choosing "Zoom In" or "Zoom Out" from the "Display" menu
- Zoom in and out by using the mouse wheel if available. The current mouse location will stay fixed, while the surrounding layout will be enlarged or reduced in size
- Press "Shift" while dragging the mouse with the right mouse button pressed will drag the layout similar to what happens in recent map service web applications

1.1.10. Global Rotation and Flip

KLayout offers an option to flip or rotate the whole window. This is a useful option for example to view mask data which a mirrored image of the chip.

To access this option, choose the desired transformation from one of the options available in the "Global Transformation" submenu of the "Display" menu.

1.1.11. Returning To A Previous View

Choose "Last State" from the "Display" menu to return to the last window shown. Choose "Next state" to switch to a more recent state again.

1.1.12. Bookmarking Views

Views (window, cell) can be bookmarked for later retrieval. Choose "Bookmark This View" from the "Bookmarks" menu. A name is required to be entered for the bookmark. The bookmark will then appear in the "Goto Bookmarks" list.

The list of bookmarks defined can be loaded or saved by using the "Load Bookmarks" and "Save Bookmarks" functions from the "Bookmarks" menu.

The bookmark list is available as a dockable tool window as well: check the "View/Bookmark List" option to enable this dock window. The bookmark list by default is shown at the bottom right side of the layout view.

Dockable Bookmark List

To navigate to a bookmark from the dockable bookmark list, double-click the entry. From the context menu (right mouse click) you can select these functions:

- **Follow Selection**: if this option is checked, the selected bookmark will immediately change the view accordingly. With this option, you can browse the bookmark list with the arrow keys while the view updates automatically.
- **Manage bookmarks**: opens the bookmark management dialog (same as from the "Bookmarks" menu).
- **Load bookmarks** and **Save bookmarks**: loads or saves the bookmarks to a file (same function as from the "Bookmarks" menu).

1.1.13. Descending Into A Cell With Context

A cell can be shown either isolated (this is the default, if the cell is the current cell), embedded (as a subcell of the current cell) or as the current cell in the context of another direct or indirect parent cell. In the latter mode, the cell is highlighted while the context cell is shown in dimmed or another, user-defined color.

To highlight a cell in a context, first choose the context cell. Then select a shape or a cell instance within the cell to show in the context and choose "Descend" from the "Display" menu or press "Ctrl+D". Now, the first child cell leading to the selected shape is highlighted, while the surrounding shapes of the parent cell (the previous current cell) is shown in dimmed colors. Choose "Descend" repeatedly to descend further into the hierarchy until the selected shape or instance is on the level of the current cell. The current cell is show underlined in the cell tree, while the context cell is shown in bold font in the cell tree as usual.

The reverse of this operation is "Ascend" (or "Ctrl+A") available from the "Display" menu.

The way how the context layout is shown can be adjusted in the setup dialog on the "Background" tab.

1.1.14. The Layer List (Layer Views)

An important concept in KLayout are the layer views. KLayout displays the layers of a layout by default in a list on the right side of the main window. This list however, does not directly reflect the layers in the layout database. Instead this list is a collection of "views". A view is a description of what is to be displayed and how.

Essentially, the entries in the layer list are pointers to layers in the database, together with a description how to paint the shapes on these layers (the "layer properties").

The pointer is the "source" of a layer view. This is typically a GDS layer and datatype, but can be a layer name (for DXF for example). There are also abstract sources (such as cell boundaries) and the sources can include selectors or modifiers. Selectors are used to only display shapes with certain user properties or from certain hierarchy levels. Modifiers transform the shapes before they are drawn for example. The source is defined by a "source specification" - this is a string describing the database layer and selectors and modifiers. A simple source string is "1/0" which is for GDS layer 1, datatype 0 without and selectors or modifiers. To change the source, use "Change Source" from the layer list's context menu.

See Transforming Views And Property Selectors for some source specification string applications and more details.

Beside the source, a layer entry has a display name. This is an arbitrary text providing a description for the user. By default - when no such name is present - the source of the layer will be displayed. To change the display name, use "Rename" from the layer list's context menu.

Plus of course, the layer views have many options to specify the drawing style, animations and decorations.

The concept of separating views from the database layers opens some interesting options:

- Layer views can refer to individual layouts from multi-layout views (through the "@1", "@2", ... notation in the source). Hence, multiple layouts can be mixed in a single layer list.
- Layers can be present in the list which do not need to be present in the database. Such a layer is shown as empty. This is important as in GDS an empty layer is equivalent to non-existing. Still you may want to have it shown in the layer list - the views offer this option.
- Vice versa, database layer may not be listed in the layer list if no corresponding layer view is present. This way, auxiliary or debug layers can be omitted from the layer list. A "wildcard specification" is available to make sure, all layers are shown if you need to see all.
- Multiple tabs can be present to provide multiple views on the same layouts. This is just an alternative set of layer views.
- Layer grouping, sorting etc. are just operations on the views, no database change is involved.

The concept on the other hand is slightly counter-intuitive at first. Here are some hints:

- Renaming a layer does not change the source - if you rename a layer to something like "1/0", you are likely to fool yourself thinking this is layer 1, datatype 0.
- Changing a layer view's source does not change the database too - it will just change the pointer. To change a layer's information in the database, use Edit/Layer/Edit Layer Specification.
- Deleting a layer from the layer list does **not delete** the layer from the database. Use Edit/Layer/Delete Layer instead.
- Additing a new layer does not immediately create the layer in the database. Only once you draw something on that layer, it is generated in the database.

1.1.15. Choosing A Layer Color

Select the layer or the layers for which to change the color and open the color chooser panel in the layer panel to the right. If the color chooser is not visible, select the small check box in the "Color" header bar. Then the color chooser is expanded.

To change the color, click on the desired color. To select a color not offered in the list, select the "More Colors" button. A color choose dialog will open.

To choose the color of the frame to draw around the shapes, without changing the fill color, use the "Frame Color" chooser panel.

Layers can be "dimmed" by making their color darker or brighter so they contrast less with the background. To do so, choose "Dark" or "Bright" from the color panel. Pressing the button multiple times makes the colors darker or brighter each time. The brightness or darkness can be reset with the "Reset" button.

1.1.16. Telling Used From Unused Layers

In some applications, the layer list will grow very large and keeping track of the important layers may be hard. KLayout provides support for that task in two ways: KLayout checks whether a layer carries any information and displays the layers in a different way in the layer list, if it is empty.

Two ways of checking the information content of a layer are provided: either a layer is said to be empty if the current cell does not have any shapes on it. Alternatively, a layer can be identified to be empty by checking if any shape is shown in the current view (more precisely if any shape's bounding box overlaps with the current view rectangle). The latter mode can be selected in the layer list's context menu with the option "Test For Shapes in View".

If a layer is determined to be empty, it is either grayed out or it is now shown at all. The latter option keeps the layer list short and is selected with "Hide Empty Layers" from the layer list's context menu.

1.1.17. Choosing A Line Style

Line styles control how the outline of the shapes is drawn. The default is a solid line. Available line styles are dotted lines, dashes lines and custom styles. To choose a certain line style, select the layer or the layers for which to change the style and choose the line style from the "Style" panel.

New styles can be created with the "Custom Style" button. A line style editor will come up that allows creating and editing of new styles. The predefined pattern cannot be changed. To apply a new style, select "More" from the style selection panel and choose the new style from the list. The custom styles are saved with the layer display properties.

1.1.18. Animating Layers

Layers can be animated, i.e. made blinking or the fill pattern scroll. Select the layer or the layers for which to change the animation style and choose the animation style from the "Animation" panel. For blinking mode, two phases can be selected: "Blink" and "/Blink". Choosing different phases for two layers makes the layers appear alternatively.

1.1.19. Changing The Display Style

The line width of the element's frame can be changed by using the width buttons on the "Style" panel after having selected the layers to apply the change on. "0px" removes the line, "1px" draws a single-pixel wide line (the default), "2px" a somewhat thicker line and so on.

"Simple" is the normal draw mode while "Marked" draws a cross on each vertex of the element. The cross size is constant so the shapes stay visible even on large scale where the elements would otherwise become single pixels.

A layer can be configured to draw a diagonal cross on rectangles. To enable this style, select "Cross" from the "Styles" toolbox. To disable it, choose "No Cross". The cross drawing will add to the fill pattern and applies to rectangles only.

1.1.20. Changing The Layer Visibility

The selected layers can be made invisible by choosing the "Hide" option on the "Visibility" panel. Choosing "Show" makes the layers visible again.

Also, double-clicking a visible layer in the layer list toggles the layer's visibility.

To make a layer "transparent" (i.e. let the other layers show through), select "Transp." on the "Visibility" panel. To make it opaque again, select "Opaque" (the default).

1.1.21. Valid And Invalid Layers

Starting with version 0.23, KLayout offers "invalid" layers: invalid layers can be visible, but don't participate in selection and snapping. They just act as some kind of background drawing.

Layers can be made invalid by choosing "Make invalid" from the layer's context menu. Layers can be made valid again by choosing "Make valid". Invalid layers are marked with a small "x" in the layer list.

1.1.22. Organizing Layers Hierarchically

Layers can be organized hierarchically. For example, certain layers can be grouped together. Choose "Group" in the context menu of the layer list (right-click the layer list). The selected layers will be replaced by a tree node that represents these layers. Click on the tree node to expand or collapse this group.

Once layers are grouped, they can be hidden or made visible with a single double-click on the node representative. The node representative also controls the appearance of the layers in the group: if a color or style is assigned to the representative, it overrides the respective style of all layers contained in the group. This way for example, the color of the layers contained in the group can be changed at once. To remove a color override of a node representative, set the color to "None".

To resolve a group, select the group representative and choose "Ungroup" from the context menu.

A variety of automatic grouping methods is provided. For example, the "Regroup views by layout index" from the layer context menu will collect all layers and put them into one group per layout shown in the panel.

1.1.23. Using Multiple Layer Setups With Tabs

With version 0.21, a new feature was introduced. Using tabs in the layer panel it is very simple to switch between different setups.

A layer tab can be created by choosing "New Tab" from the "Tabs" submenu in the layer panel's context menu (right mouse button click). A new tab will appear at the top of the layer properties panel. Initially this tab will be a copy of the current setup. Any edits on the layer properties will apply to this tab only. When switching to a different tab, the layout view will reflect the new tab's settings. That way, different setups can be prepared and easily exchanged.

When the layer properties are saved, the layer properties file will contain all tabs. Thus, a multi-page setup can easily be stored and retrieved.

The initial title of the tab will be the tab number. The title can be set with the "Rename Tab" function in the "Tabs" submenu of the layer panel's context menu. To remove a tab, choose "Remove Tab".

1.1.24. Removing And Adding Layers To The Layer Set

The layers shown in the layer list are rather "pointers" to the actual layout layers and not representing the actual layers. Because of this, these layers are more precisely referred to as "views". Layers can be removed and created again without affecting the actual layout data.

To create a layer, choose "Insert View" from the layer context menu (right mouse button click on the layer list). Then, an input dialog prompts for the source specification. The source specification tells, from which actual data layer to take the displayed data from. The most simple form of a source specification is "layer/datatype" (i.e. "5/0") or the layer name, if an OASIS layer name is present. This specification can be enhanced by a layout index. The first layout loaded in the panel is referred to which "@1" or by omitting this specification. The source specification "10/5@2" therefore refers to layer 10, datatype 5 of the second layout loaded in the panel.

Source specifications can be wildcarded. That means, either layer, datatype or layout index can be specified by "*". In this case, such a layer must be contained in a group and the group parent must provide the missing specifications. For example, if a layer is specified "10/*" and the parent is specified "*/5", the effective layer looked for will be "10/5". Unlike the behaviour for the display styles, the children override (or specialize) the parent's definition in the case of the source specification.

The layer list can be cleaned up to remove layer views that do not correspond to actual layout layers using the function "Clean up views" from the context menu. Similar, layers that are present in the layout but for which there is no view can be added using the "Add other views" method.

1.1.25. Transforming Views And Property Selectors

The source specification described in the section before is much more powerful than just allowing to describe the data source. In addition to that, the layer can be geometrically transformed and the display can be confined to shapes that belong to a certain class described by a property selector.

A geometrical transformation is specified by appending a transformation in round brackets to the layer/datatype source specification. The format of this transformation is (in any order):

```
( [<dx>,<dy>] [r<angle> | m<angle>] [*<mag>] )
```

For example, "(r90)" specifies a rotation by 90 degree counter-clockwise. "(0,100.0 m45 *0.5)" will shrink the layout to half the size, flip at the 45 degree-axis (swap x and y axes) and finally shift the layout by 100 micron upwards.

A detailed explanation of the transformation syntax can be found in Transformations in KLayout.

Transformations accumulate over the layer hierarchy. This means, that if a layer is transformed and the layer is inside a group whose representative specifies a transformation as well, the resulting transformation is the combination of the layer's transformation (first applied) and the group representative's transformation.

Multiple transformations can be present. In this case, the layout is shown in multiple instances.

A particular application is to regroup layers by layout index and assign a transformation to the group representative belonging to a certain layout such that the layouts get aligned.

The property selector is specified in square brackets. A selector combines several expressions of the form "<property>==value>" or "<property>!=<value>" with operators "&&", "||", "!" and allows usage to round brackets to prioritize the evaluation of these operators:

```
[ <expr> ]
```

In GDS2 files, the property is always named with a integer value which is written with a single hash characters (i.e. "#43". The value of a GDS property is always a string. A string is either written as a text atom or can be enclosed in single or double quotes. The following is an example for a valid property selector for GDS files:

```
10/5 [#43==X]
```

With this source specification, the layer will show all shapes from layer 10, datatype 5 which have a user property with number 43 and value string "X". A more complex example is this:

```
10/5 [!(#43==X&&(#2==Y||#2==U))]
```

With OASIS files, the properties can be named with a string. In this case, the property selector can be "[prop==X]" for example. In addition, the value can be a an integer or a double value. This is reflected by the choice of the value: "[prop==#200]" will check, if the property named "prop" has an integer value which is 200. In the same fashion, "[prop==##0.5]" checks, if the property "prop" has a double value and this is 0.5.

Property selectors combine over a layer hierarchy. This means, that if a group representative specifies a property selector and a layer in this group specifies a selector as well, only those shapes will be shown that meet both criteria.

A general description for the source notation is found here: About Layer Sources.

1.1.26. Specifying Explicit Hierarchy Levels For One Layer Or A Layer Group

By default, only the hierarchy levels that are selected in the hierarchy level selection boxes are shown, i.e. if levels 0 to 1 are selected, just the top level shapes and instances are shown. This selection can be modified for certain layers or layer groups. To specify a different hierarchy selection for a certain layer, use an optional source specification element, the hierarchy level selector:

#[<lower-level>][`..` <upper-level>]

Upper and lower level can be omitted. In this case, the respective level is not overridden. The upper level can be '*' which means: every level that is available. If just one level and no ".." is given, it is taken as upper level and the lower level is set to zero.

Some examples might illustrate this:

`#*`	Display all hierarchy levels
`#0..1`	Display top level only
`#..5`	Override upper level with 5
`#2..`	Override lower level with 2
`#..*`	Override upper level setting by "all levels"

Modifications of this notation are provide in order to support more use cases. Instead of specifying a single number for the level, the following alternative notations are supported:

`(1)`	Relative specification: Hierarchy level 1 related to the current cell's level. The effective specification differs in "Descend" mode where the current cell is on a lower hierarchy level than the context cell which is the top cell drawn.
`<1`	Constrained specification: Hierarchy level 1 or less if the upper or lower default level set in the user interface is less.
`>1`	Constrained specification: Hierarchy level 1 or greater if the upper or lower default level set in the user interface is greater.
`(>1)`	Combined specification: Hierarchy level 1 related to the current cell's level or less.
`>*`	Equals the currently set maximum hierarchy level.

For example:

`#(0)..(1)`	The top level of the current cell (works also in "Descend" mode).
`#>0..<1`	Everything exactly on top level unless the top level is not selected in the controls.
`#>1..<*`	Everything below the context cell's top level unless not selected by the user interface controls.
`#(>1)..<*`	Same than before but related to the current cell, not the context cell.

1.1.27. Loading And Saving The Layer Sets

The visual layer properties can be saved to a file using the "Save Layer Properties" function from the "File" menu. This list can be loaded again using the "Load Layer Properties" function.

1.1.28. Creating A Screenshot

To save the canvas as a PNG file, choose "Screenshot" from the "File" menu or press the "Print" key. A file dialog box will appear in which the file can be specified where the screenshot is saved to.

1.1.29. Doing Measurements

A measurement can be performed by clicking on the ruler icon in the toolbar and selecting "Ruler" from the drop-down options. Left-click on a point in the layout and then left-click again to specify the second point. A ruler will be shown that indicates the distance measured.

A more convenient way is provided with the single-click measurement ruler. Select "Measure" from the drop-down options of the ruler symbol. In this mode, a single click will set a ruler to the specified position. This feature will look for edges in the vicinity of the ruler and set the ruler to connect the neighboring edges. The ruler is attached perpendicular to the edge next to the initial point.

You can mark a position with a single click by selecting the "Cross" ruler type. Clicking at a location will place such a ruler. The ruler shows the x and y coordinate.

The "Multi-Ruler" allows concatenating multiple rulers into a single object. Click and the first point to start such a ruler. Then click on more points to add new segments to the ruler. Each segment is shown as an individual ruler with tick marks and a length. Finish the sequence with a double-click.

The "Angle" ruler allows angle measurements. Three clicks are required to define a "V" like arrangement of two rulers. The angle enclosed by the two lines forming the "V" is shown in the ruler.

Another special ruler, the "Radius" ruler is also a three-click type. Specify three points to define a circle. The ruler shows the radius of this circle and the circle outline.

Rulers can be configured in manifold ways. Use "Rulers And Annotations Setup" in the "Edit" menu to open the ruler configuration dialog. A ruler can be made to snap to edges of objects by selecting "Snap to edge/vertex". Ruler orientations can be constrained by using the "Angle Constraint" options. The number of rulers can be limited using the "Limit number of annotations" setting.

While drawing or moving one point of a ruler, the direction constraint can be overridden with the Shift and Ctrl keys: pressing Shift while moving the mouse will enforce orthogonal constraint, Ctrl will enforce diagonal constraint while pressing both will release any direction constraint.

All rulers can be cleared using the "Clear all rulers" function from the "Edit" menu.

Ruler dragging can be canceled with the "Esc" key or using the "Cancel" function from the "Edit" menu.

Rulers can be moved by selecting "Move" mode with the speedbar buttons in the toolbar or "Move" from the "Mode" sub-menu in the "Edit" menu. Then left-click and drag the ruler or the ruler end point that should be changed.

Rulers can be deleted selectively by selecting a ruler in "Select" mode and pressing "Delete".

Rulers can be modified in a variety of ways. For example, rulers can be shown as arrows. To edit the properties of a ruler, double-click the ruler or select it and use "Properties" from the "Edit" menu. See Ruler Properties for a description of the properties.

Multiple templates can be configured to be available for rulers. Each template defined will be shown in the "Ruler" mode toolbar button's drop-down menu. If a template is selected, new rulers produced from this template will inherit the template's properties. Templates are managed in the ruler setup page ("Setup" from the "File" menu) or "Ruler And Annotation Setup" from the "Edit" menu.

1.1.30. Ruler Properties

These are the properties that can be configured for rulers:

- **Labels:** depending on the outline of the ruler, up to three labels an be present. Each label can be configure individually to either show a text or the measurement values. The main label is always present, X and Y labels are only present, if the ruler has an explicit vertical or horizontal component (all outline styles except "diagonal"). For the main label the position of the label can be specified ("P" setting): the label can be made to appear on the first or the second point or in the middle of the ruler. The Alignment of the labels can be specified too: whether the appear left or right-aligned or centered.
- **Style:** the style determines how the ruler or its components are drawn. This can be "ruler-like" (with ticks), arrow style, a plain line or with cross markers at the end.
- **Outline:** the outline determines how the two points forming the ruler are connected to render the ruler shape. This is either just one line ("diagonal"), a horizontal and a vertical line (in some outline styles combined with the diagonal line) or a box given by the two points of the ruler. A special outline is the ellipse which draws an ellipse inside the box defined by the ruler.
- **Angle constraint:** the orientation of the ruler can be restricted in several ways, i.e. just being horizontal. By default, the ruler uses the global setting. It can however be configured to provide its own constraint.
- **Object snapping:** each ruler can be configure to snap to the closest object edge or vertex. By default, the rulers use the global setting. It may be disabled however for each ruler.
- **Mode:** in normal mode, two clicks are required to define a ruler: to set the first point and to set the second one. In "Single click" mode, a single click will set both points to the same. In "Auto measure" mode, the points will be determined by looking for edges in the vicinity of the click point and adjusting the points accordingly.

The "Label format" is an arbitrary text with embedded expressions that may represent a measurement value. Each such expression starts with a dollar sign, followed by the expression string. The expression syntax supportis the basic operations ("*", "/", "+", "-" ..), bitwise operations ("|", "&", ..), the conditional operator ("x:y?z") as well as some functions, i.e. "abs", "sqrt", "exp". It includes a "sprintf" function. These are some examples:

- **$X**: The value of the X variable (the horizontal distance, see below for a complete list of variables).
- **$(sprintf('%.2f',X))**: The value of the 'X' variable formatted as two digit fixed precision value.
- **$(abs(X)+abs(Y))**: The manhattan distance of the ruler.
- **$min(X,Y)**: The minimum of X and Y.

A description of the expression syntax and the functions available can be found in About Expressions.

This is a list of all variables available:

- **D:** The length of the ruler in micron units.
- **L:** The manhattan length of the ruler in micron units.
- **U:** The x-position of the ruler's first point in micron units.
- **V:** The y-position of the ruler's first point in micron units.
- **P:** The x-position of the ruler's second point in micron units.
- **Q:** The y-position of the ruler's second point in micron units.
- **X:** The horizontal extension of the ruler in micron units.
- **Y:** The vertical extension of the ruler in micron units.
- **A:** The area enclosed by the ruler (if it was a box) in square millimeters.

- **G:** The angle enclosed by the first and last segment of the ruler (used for angle measurement rulers).

1.1.31. Adding Images

For some applications it is necessary to show flat pixel data together with the layout. That can either be a SEM image taken or some output of a simulation tool. KLayout provides a way to add images to the display and show them below the drawn layout.

Currently, images can be read from any commonly used image format available in Qt (i.e. PNG, JPG, TIF ...). Color and monochrome images are supported. Internally an image is stored as a matrix of float values and it is possible to write custom importers using RBA.

To add an image, use the "Add Image" function from the "Edit" menu. An image property dialog will appear where the image can be specified. Choose an image using the "Browse" button next to the file name box.

An image has a variety of properties which mainly affect the way it is displayed:

- **Pixel size:** The size of one pixel in micron units. This affects the total size of the image.
- **Center:** This is the point where the center of the image is placed (in micron units).
- **Rotation:** An arbitrary angle by which the image is rotated.
- **Shear/Perspective Tilt:** shear and perspective tilt angles. Although it is possible to specify these angles explicitly it is far easiert to use the landmark adjustment feature to align an image with a layout.
- **Mirror flag:** If this option is checked, the image is mirrored at the bottom edge before it is rotated.
- **Pixel value range:** The pixel value corresponding to minimum and maximum. For normal 8 bit image formats, these values are 0 and 255. They can be adjusted which allows brighten or darken images. For float images (i.e. simulation data), this value should reflect the bounds of the output values, i.e. 0.0 and 1.0 for normalized data.
- **Color mapping:** For monochrome images, the values are converted to colors with a mapping function. The image properties page contains a tab for specifying an arbitrary mapping of data values to colors. This is achieved by placing color sample points on the data range axis and assigning colors to them. Double click at the axis to set new points, click on them to select them and adjust their color with the color box. Select and press "Del" to delete a sample point.
- **Brightness, Contrast and Gamma:** Three sliders for changing these values are provided on the respective tab.
- **RGB channel gains:** Additionally, each color channel can be weighted with a given factor on the respective tab.

Once an image is placed, it can be moved and resized using the "Move" function. The images properties can be adjusted using the "Properties" function from the Edit menu or double-clicking at the image.

With KLayout 0.22, it is possible to define landmarks which can be set at arbitrary positions in the image and aligned with corresponding layout features by dragging them to the desired target location. See Using Landmarks To Align Images for details.

An arbitrary number of images can be placed on the layout view. To store the setup, save the session using the "File/Save Session" function.

1.1.32. Using Landmarks To Align Images

"Landmarks" are arbitrary positions in the image. You can define such positions graphically in an image and then, within the layout view, use the defined landmarks as handles to align an image with a layout. Depending on the number of landmarks defined, you can use this feature to compensate the shift of an image, rotation, shear or even perspective distortion.

Landmarks are defined in the landmark editor. Open the landmark editor by pressing the "Define" button in the image properties dialogs.

The landmarks editor shows the image in the left panel and a list of the landmarks already defined int the right panel. You can zoom around the image view using the same methods than in KLayout's layout view (mouse wheel, middle mouse button, zoom box with right mouse button).

To add a landmark choose "Add" mode. Then click at the desired position of the new landmark. It will be shown in the image as a small target cross symbol. To delete a landmark, choose "Delete" mode and click on the landmark to delete. You can also select a landmark from the list and enter Delete mode. In that case, the currently selected landmarks are deleted.

To move a landmark, enter "Move" mode and move the landmark. Please note that the move operations follows KLayouts "pick and place" philosophy, so you have to click with the mouse twice: once to pick the landmark and second to drop it. The advantage of this approach is that you can zoom while dragging the landmark.

It does not make much sense to define more than 4 landmarks because for more the transformation derived from the landmarks is overdetermined. For best accuracy it is important to select landmark positions that span a large region. In particular, landmarks should not be too close and three landmarks should not form a line.

Dependent on the number of landmarks, KLayout can adjust the following parameters of the image:

- **1 Landmark**: displacement
- **2 Landmarks**: displacement, rotation and magnification
- **3 Landmarks**: displacement, rotation, magnification and shear
- **4 Landmarks**: displacement, rotation, magnification, shear and perspective distortion

Once the landmarks are defined, they can be used to adjust the image's display transformation. To do so, enter "Move" move in KLayout. When the mouse is over the image, the landmark symbols are shown and can be picked with the mouse. When you pick up a landmark and move the mouse, KLayout will adjust the image transformation and display the image frame according to the new transformation. When you drop the landmark, the image will be transformed accordingly.

Depending on the number of landmarks, KLayout can adjust either some or all transformation parameters such as displacement, angle etc. such that the other landmarks stay on the same position. In some cases that may not be desired. For example, if four landmarks are defined and one of them is moved, the result may be an extreme perspective distortion. Usually one would do a coarse adjustment by roughly adjusting the position and magnification and use the perspective transformation only to fine-adjust the image finally.

Hence, KLayout allows reduction of the degree of freedom it has when adjusting the transformation. While you drag a landmark, you can use these modifier keys:

- **Shift**: Only adjust displacement
- **Ctrl**: Adjust displacement, rotation and magnification
- **Shift+Ctrl**: Adjust displacement, rotation, magnification and shear

1.1.33. Browsing Shapes

A simple shape browser allows browsing all shapes on a layer. To do so, select the layer or layers to browse in the layer list and choose "Browse shapes" from the "Tools" menu.

A browser dialog will appear that lists the cells, shapes and cell instances. Selecting a cell will display all shapes in the cell in the middle list and the cell's instances with respect to the top cell in the right list.

If a shape is selected, the layout canvas highlights this shape by drawing a marker box around the shape and zooming to the shape. How the shape is shown can be configured on the "Configure" tab of the shape browser dialog or on the respective page in the "Setup" dialog.

Another feature that among many other things allows inspection of shapes is the "Search and replace" function (see Search and Replace).

1.1.34. Browsing Instances

All instances of a cell can be browsed by selecting the cell in the cell list (not making it the new top), and choosing "Browse instances" from the "Tools" menu. A simple instance browser comes up that shows all cells that the given cell is instantiated in and how the cell is instantiated.

Another feature that among many other things allows inspection of instances is the "Search and replace" function (see Search and Replace).

1.1.35. The Marker Browser

KLayout offers a generic concept of storing error markers or related information. This concept is called the "Report database" (RDB). An arbitrary number of report databases can be associated with a layout view. Usually, each database refers to a certain layout but that is not a strict requirement.

A report database primarily is a generic collection of "values", which can be strings or other items. Usually, a value is a collection of geometrical objects which somehow flag some position or drawn geometry. Multiple of such values comprise a "marker item". The report database associates these marker items with additional information:

- **Tags:** Flags that indicate certain conditions. The marker browser uses a couple of predefined tags like "important", "waived" and "visited" which can be set or reset by the user indicating whether a marker item is considered important or an error has been waived.
- **Image:** A marker can be assigned a screenshot image which serves for documentation purposes.

Marker items are organized into categories. Each marker item must be associated with a category. Categories themselves can be organized hierarchically, i.e. categories can be split into sub-categories. This offers a way of improving the organisation of such categories.

Marker items are usually associated with a cell, i.e. where a error was detected. By default, a marker item is simply associated with the top cell.

The report database uses a proprietary format based on XML which is capable of storing the annotations provided by the database. It is possible however to import Calibre DRC ASCII format files.

The marker browser is a tool to browse a report database associated with a view. The marker browser can be started using the "Marker Browser" function in the "Tools" menu. The marker browser tracks whether a marker has already been visited similar to the "read" flag in a mail client. This allows tracking of a review session. The "visited" state is reflected in the database file.

In the marker browser, use the "Open" button to load a XML database file or import files from other formats. Choose "Reload" to reload a file and "Save As" to write a database in XML format.

The marker browser offers three panels:

- **Directory:** This panel lists the categories and cells of the database. Categories or cells with unvisited markers will be shown in bold font. Such with no markers at all are shown in green color. It is possible to suppress these categories or cells by deselecting "Show All" in the directory's context menu. To have the lists sorted by marker count, click at the header of the count column.
  Multiple categories or cells can be selected. In that case, the markers panel will show the markers of all selected cells or categories.
- **Markers:** This panel lists the markers in the selected category and/or cell. A length of the list is limited and can be changed on the configuration page ("Configure" button on the marker browser or in the setup dialog). Various tags are shown in this panel as well. The list can be sorted in various ways by clicking at the respective header.
  When a marker is selected in this list, it will be highlighted in the layout, provided a suitable layout is associated. The way a marker is highlighted and how the view is adjusted can be specified on the configuration page.
- **Info:** This panel summarizes the information for the selected marker. If a screenshot was associated with the marker it is shown here. Click on the thumbnail image to show it in a separate window in the original size.

Similar to the shape and instance browsers, the marker browser offers navigation buttons to select the next marker, category or cell.

The marker browser supports "tagged values": each marker can be associated with a value that has a name. Such values can be imported from RVE files where they are called properties. Tagged values are generated by certain generators and can represent measurement values or similar. When tagged values are present, they will be shown in the markers list. The markers can be sorted by these values by clicking on the appropriate header.

1.1.36. Technology Management

A new feature of KLayout 0.22 is technology management. Technology management summarizes features which require a certain interpretation of a layout. In particular, layout layers are assigned a physical meaning, for example via layers or active area layers in CMOS technologies. Since that interpretation often is depending on the technology the product will be fabricated with, the ability to provide multiple setups is summarized as "technology management".

The "Technology Manager" in the "Tools" menu is the user interface that allows creating, deleting and editing of technology setups. Read more about the technology manager in About Technology Management

1.1.37. Selecting Rulers, Shapes Or Instances

Rulers, images, shapes or instances can be selected by either clicking on the shape, instance, ruler or image in "Select" mode or by dragging a selection rectangle with the left mouse button pressed. In this case, all objects inside the selection rectangle will be selected.

Pressing the Shift key in addition to selecting shapes, instances or other objects will extend the current selection. Pressing Ctrl key will remove the given objects from the selection.

For layout objects (shapes and instances), it is possible to select the objects through the hierarchy or on the level of the current cell only (top level objects only). By default, objects are selected through the hierarchy. That means, objects in child cells can be selected. By extending the selection over multiple instances of a cell, the same object can be selected multiple times. While editing objects inside a child cell may be a desirable feature, this can lead to confusing effects in the end. Hence, it may to more intuitive to only work on top level, i.e. on the current cell. To enable this mode, check the "Select Top Level Objects Only" menu item in the "View" menu or the same in the edit mode options dialog (F3).

In top-level only mode, instances of cells are selected only if the cells have a visual appearance. This is the case, if they contain at least one shape on a visible layer for example. If they are empty with respect to the visible layers, they cannot be selected. If the cell's box is shown, because the hierarchy levels shown are confined a sufficiently small interval, the cell instances will always be selected.

Images and rulers are simple objects not embedded in a hierarchy and can be selected by clicking at them or enclosing them in the selection box.

Object properties can be inspected or edited (if allowed) by opening the properties dialog. To edit or inspect the properties of selected objects, choose "Properties" from the "Edit" menu. A dialog will open which shows the properties for the first selected object. To proceed to the next object push the "Next" button, to go back to the previous object push the "Previous" button.

Depending on the application mode, the properties of the selected objects may be edited too. To apply changes to the current object, choose "Apply". To apply the changes and close the dialog, choose "Ok". To apply the changes to the current object and all other selected objects of the same kind, choose "Apply To All" (if applicable). To close the dialog without applying any changes, choose "Cancel".

"Apply To All" will try to apply the changes in some smart way to other objects. For example, for boxes, and change of the box dimensions will be applied in the same way (same shift in the four directions) than for the current object. If the width of a path has been changed, the same width will be applied to all other paths and so forth.

For shapes and instances, the "User properties" can be edited too. "User properties" are arbitrary properties attached to shapes and instances. They consist of a key (preferably a number for GDS2 compatibility) and a value (preferably a string in GDS2). Multiple properties of that kind can be attached to one object. User properties can be edited by pushing the "User Properties" button which will open a dialog that allows editing of the properties. If this dialog is closed with "Ok", the user properties will be kept, but applied to the layout object only after pressing "Apply"/"Ok" or "Apply To All". The latter will apply the new properties to all other shapes of the same kind.

Finally, for shapes and instances, the "Instantiation" button will show the instantiation path of the shapes or instances.

The layer of a shape cannot be changed through the properties dialog. To move a shape to a different layer, use "Change Layer" from the "Selection" sub-menu in "Edit".

1.1.38. More Configuration Options

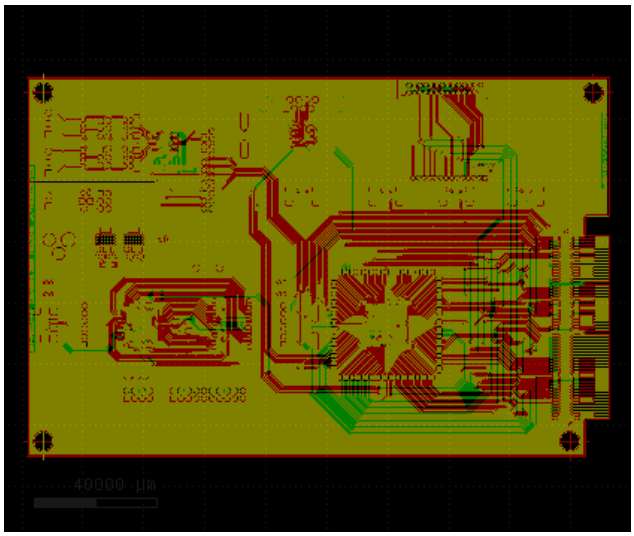
The option dialog available with the "Setup" function from the "File" menu offers numerous configuration options from background to rulers configuration.

In this dialog for example, the color palette can be edited, so that different colors are available or the stipple palette can be configured. In addition, it is possible to define the order how these colors or stipples are assigned to layers initially and which colors are not used for layer coloring.

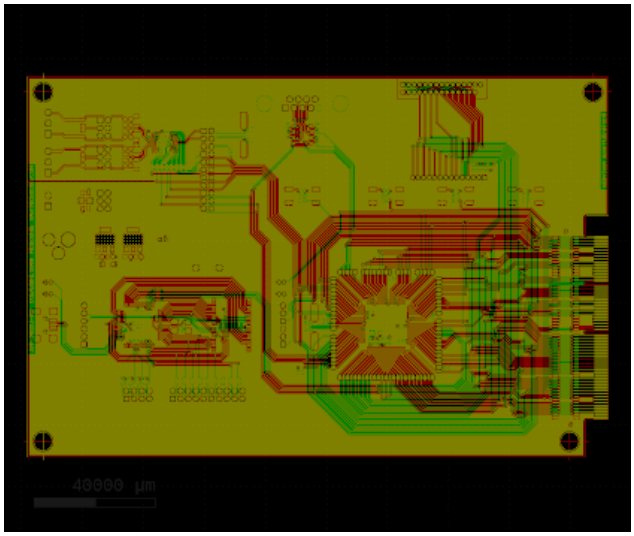
A particular useful feature is the oversampling scheme. Oversampling is provided as an option to enhance the image quality. The image is rendered at a higher resolution and then downsampled to the screen resolution. In effect, lines appear thinner and more details can be resolved. As a negative side effect currently the stipple pattern becomes finer and the crosses in marker mode are smaller. On the other hand the resolution effect can be quite impressive.

Oversampling can be enabled on the "Display/General" page in the setup dialog. 2x and 3x oversampling is provided. The following screenshots illustrate the effect of oversampling:

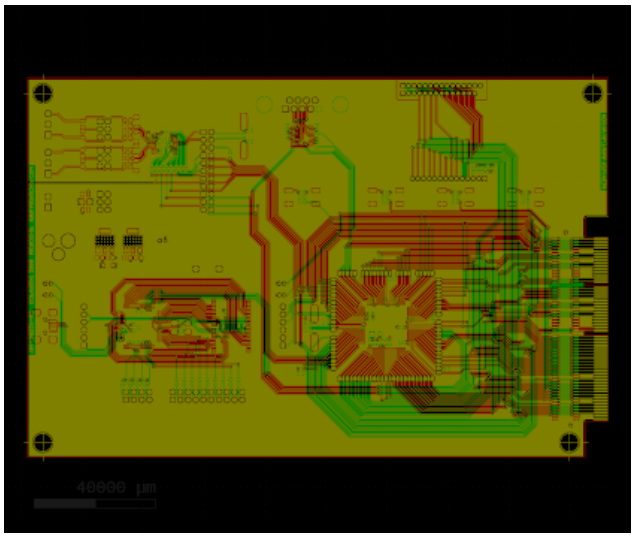
Normal (1x)



2x oversampling



3x oversampling



1.1.39. Undo And Redo

Most operations such as changing of layer colors can be undone using the "Undo" function from the "Edit" menu. Analogous, the operations can be redone again using the "Redo" function from the "Edit" menu.

1.1.40. Saving A Layout Or Parts Of It

A layout or a subcell of it can be saved to either GDS2 or OASIS. To save a layout, choose "Save As" from the "File" menu. To save just a cell, select the cell in the cell tree (it does not need to be the currently shown one) and select "Save Current Cell As" from the context menu (right mouse button) of the cell tree.

A file dialog will pop up to select the file name to which to write the cell or layout. After a file has been selected, an option dialog will be shown to specify further options. In this dialog, it is possible to constrain saving to a subset of layers, i.e. just visible ones. Also, the database unit can be changed or the layout can be scaled by a given factor.

For OASIS, a compression level can be specified. At a level of 0, no particular attempt is made to compress shapes. At higher levels, shapes are classified and array compression is tried. The higher the level, the more attempts are made to compress shapes into arrays. In particular for flat layouts, compression of shapes requires some memory and slows down OASIS writing considerably.

1.1.41. Saving And Restoring A Session

A session can be saved and restored later. A "session" involves the files loaded, bookmarks, annotations, layer settings, hierarchy settings, images and application setup. Sessions are stored as XML files with the suffix ".lys".

To save a session, choose "Save Session" from the "File" menu. To restore a session, choose "Restore Session". KLayout can be started with a certain session using the "-u" option from the command line followed by the session file. On Windows installations, session files are registered as being opened automatically by KLayout.

1.1.42. Further View Options

The "View" menu offers some more options to tune the display:

- **Show Grid**: This is a fast access method for the respective configuration option. Unchecking this menu entry hides the grid lines and the scale bar.
- **Grids**: this submenu allows fast access to the current grid setting. You can choose from one of the default grids that are configured on the "Default Grids" page in the "Application" section of the setup dialog. Grids are mainly used for editing geometry but also for snapping rulers if they are configured to do so.
- **Show Texts**: when this option is unchecked, texts are not drawn. That applies to text objects and user properties. Drawing is somewhat faster when texts are not drawn.
- **Show Cell Frames**: when this option is unchecked, cell frames from hidden cells or cell below the hierarchy levels selected are not drawn.
- **Show Layers Without Fill**: when this option is checked, the shapes are drawn without fill. Hence the layout is shown in a kind of wire frame. This option is intended to temporarily disabling the fill pattern. Keeping this option checked may be confusing because setting a fill pattern may not have any effect then.
- **Synchronized Views**: with this option, all views (tabs) in the main window are synchronized, i.e. they show the same region of the layout. This option can be useful if two layouts are loaded in different views for clarity, but identical regions need to be inspected.
- **Select Top Level Objects**: this menu item gives a quick access to the respective editor option. If this option is checked, only objects from the top level are selected. This is in particular important for editing because editing a subcell can cause effects in other places there that cell is instantiated but no edit is intended.
- **Toolbar**, **Navigator**, ...: disable or enable the respective tool windows.
- **Highlight Object Under Mouse**: with this option checked, the object under the mouse is highlighted temporarily if the mouse hovers over that object. Since that can be confusing in some (rare) cases, this menu item provides quick access to that configuration option.

1.2. Editing Functions

Welcome to KLayout's user manual. This is the manual chapter covering the editing features of KLayout. The editor features are available only if KLayout is started in editor mode. The following subtopics are available:

1.2.1. Edit Mode

Editor functions can only be used if KLayout runs in edit mode. KLayout can be put into editing mode by simply supplying the "-e" option on the command line:

`klayout -e` [<input file>] [`-l` <layer properties file>]

Accordingly, with the command line option "-ne", non-editable mode can be enforced.

On Windows, there are start menu entries for editor and viewer mode. Klayout can be configured to use editing mode as default when started. To enable editing by default, check the "Edit mode" check box on the "Application" tab in the setup dialog ("File/Setup").

In editing mode, some optimizations are disabled. This results in somewhat longer loading times and a somewhat higher memory consumption. The actual increase strongly depends on the nature of the input file: for example, OASIS shape arrays are not kept as such in editing mode and resolved into individual shapes.

1.2.2. Basic Principles Of Editor Mode

This section covers the basic working principles of editor mode.

1.2.2.1. Pick And Drop Principle

Most drawing programs employ the click-and-drag paradigm: left-click on an element and drag it to the destination keeping the mouse button pressed. Although being pretty intuitive, this principle has one disadvantage: it is hard to do something other than dragging, while you keep the mouse button pressed. In particular this means: no zooming (or would you like to press the right mouse button as well, draw the zoom box and then release just the right mouse button ...?). In order to allow zoom and potentially other operations, KLayout employs the pick-and-drop-principle.

In pick-and-drop, you pick an element by clicking at it with the left mouse button, move it (without any mouse button pressed) and drop it (by left-clicking at the target position). Since the mouse button is not pressed, the mouse is free for other operations: just the dragged item is "sticking" to the mouse cursor.

In addition, while dragging the object, Shift and Ctrl keys can be used to force certain direction constraints or override the ones specified in the options (i.e. "move" or "edit" options): The Shift key forces KLayout into orthogonal mode: movements are restricted to horizontal or vertical unless not applicable. The Ctrl key forces KLayout into diagonal mode: movements are restricted to horizontal, vertical or the diagonal axes. Ctrl plus Shift will release all directional constraints - movements will be allowed in any direction.

1.2.2.2. Basic Editor Options

Most tools being using in editing mode have certain options, i.e. when drawing a path, the width and extension mode has to be specified. There exists a general setup dialog for editing options. It can be opened using "Editor Options" from the "Edit" menu or using the F3 shortcut (unless overridden).

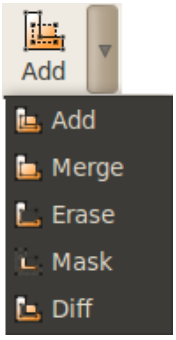
In the dialog there is always a generic settings tab and - depending on the tool chosen - a tool specific tab. On the generic tab, these settings can be changed:

- **Editor grid:** Every editing operation is confined to that grid. It can be either disabled, aligned with the global grid (used i.e. for rulers and display) or specified explicitly. It can even be anisotropic - i.e. there can be a different grid in y than in x direction.
- **Connection angle constraint:** When a connection is drawing, i.e. a segment of a path or an edge of a polygon, this mode determines, if the segment or edge is confined to certain directions. In "Any Angle" mode, there is no such confinement. In "Diagonal" mode, the edge or segment can be vertical, horizontal or in one of the two diagonal directions. In "Manhattan" mode, only horizontal and vertical edges or segments are allowed.
- **Movement direction constraint:** When something a dragged (i.e. moved), this mode determines if the movement is confined to certain directions. In "Any Direction" mode, there is no such confinement. In "Diagonal" mode, the movement can be vertical, horizontal or in one of the two diagonal directions. In "Manhattan" mode, only horizontal and vertical movements are allowed.
- **Top level selection mode:** In top level selection mode (when the check box is checked), only elements on the level of the currently shown cell are selected ("top level" refers to the top level of the currently shown cell here). That means, that if shapes from a subcell are selected, the whole instance of this subcell is selected.
  In hierarchical selection mode (when the check box is not checked), elements are selected from subcells are well. This mode allows in-place editing of subcells which is a powerful feature but also creates strange side effects if other instances of this cell changes as well.

Whenever you change something in the settings dialog, use "Apply" or "Ok" to apply your changes.

1.2.2.3. Background combination modes

KLayout offers several ways to combine the shapes drawn with shapes that are already there. The mode can be selected with the "Select background combination mode" tool button in the tool bar:



These are the modes available:

- **Add**: this is the default mode: the drawn shape will simply be added to the existing shapes.
- **Merge**: in this mode, the drawn shape will be merged with the existing shapes. This operation will always render a polygon that is the drawn shape merged with any polygons that touch or overlap the drawn shape. Paths will be converted to polygons always. This mode is equivalent to a boolean "OR" operation.
- **Erase**: in this mode, the drawn shape will be subtracted from the existing shapes. This mode can be used to create notches or slits in shapes touching or overlapping the drawn shape. The drawn shape will vanish. This mode is equivalent to a boolean "NOT" operation.
- **Mask**: in this mode, the drawn shape will act as a mask for existing shapes. Only the parts overlapping the drawn shape will remain. This mode is equivalent to a boolean "AND" operation.
- **Diff**: finally, in this mode, the drawn shape will invert existing shape. This mode is equivalent to a boolean "XOR" operation.

1.2.2.4. Selection

The basic entity that some operations work with is the "selection". This is basically a set of shapes of instances on which an operation should be applied. A selection can be established by either clicking on a element in "Select" mode or by dragging a selection rectangle. When the mouse is released, all elements inside the selection rectangle are selected.

The selection set can be modified by adding elements (press the Shift button in addition to selecting elements), by removing elements (press Ctrl in addition) or by toggling the selecting (press Shift and Ctrl in addition: remove already selected ones and add new ones).

1.2.2.5. Partial Mode

"Partial editing" is a powerful feature that allows modification of shapes. Edges or segments of polygons resp. paths can be moved, vertices, edges or segments from polygons or paths can be deleted and new points can be inserted into polygons and paths. "Partial editing" can be applied to a complex partial selection: Multiple edges or vertices can be selected and deleted or moved.

The normal selection works "full element". In this mode, the whole shape is being moved or deleted. Only in full element mode, shapes or instances can be sent to the clipboard.

1.2.3. Basic Editing Operations

This section covers some basic operations when editing layout.

- Creating A Layout From Scratch
- Creating A New Layer
- Creating A New Cell
- Creating A Polygon
- Creating A Box
- Creating A Text Object
- Creating A Cell Instance
- Moving The Selection
- Other Transformations Of The Selection
- Partial Editing
- Moving Shapes To A Different Layer
- Other Layer Operations
- Copy And Paste Of The Selection
- Delete A Cell
- Rename A Cell
- Copy And Paste Of Cells

1.2.3.1. Creating A Layout From Scratch

To start with a fresh, empty layout, choose "New" from the "File" menu. A form is opened that requires you to specify some basic parameters. These are:

- **Top cell:** this is the name of the first (and only) cell that will be present in the layout.
- **Database unit:** this is the database unit (the conversion factor between integer coordinates and micron units. This is basically the "resolution" of the layout.
- **Initial window size:** this is the size of the initial window shown, when the top cell is opened the first time. Since the initial view is empty, there is no geometrical guidance. By specifying an initial size, at least the "canvas" dimensions are known.

If a default layer properties file is specified in the setup dialog ("Application" tab), this is loaded into the layer view list automatically. Without such a file, the layer list is empty at the beginning and layers must be created with "Layer/New" from the "Edit" menu, before any shapes can be drawn.

1.2.3.2. Creating A New Layer

You can create new layers using the "Layer/New Layer" function from the "Edit" menu. You are prompted to enter GDS layer and datatype numbers and optionally an OASIS layer name. On "Ok", the layer will be created and will be inserted into the layer panel.

1.2.3.3. Creating A New Cell

You can create new cells using the "New Cell" function from the hierarchy panel's context menu (right mouse click on the cell hierarchy panel). You are prompted to enter the new cell's name (a cell with that name must not exists yet) and a window size. The window size is the initial dimension of the view when the new cell is shown. Apart from that it does not have any meaning.

1.2.3.4. Creating A Polygon

Select "Polygon" mode from the toolbar. Choose a layer from the layer panel in which to create a new polygon. Left-click at the first vertex of the polygon. Move the mouse to the next vertex and place a new one with a left mouse button click. Move to the next vertex. Depending on the connection mode, the edges created are confined to certain directions. See Basic Editor Options for a detailed description of the modes. Use the "editor options" dialog (F3 shortcut) to change the mode, even during editing.

Double-click at the final point to finish the polygon. Press the ESC key to cancel the operation.

A polygon will never be "open": there are always edges connecting the current vertex with the initial one. Depending on the mode, this final connection is either a straight line or a combination of edges. In "diagonal mode", there are manifold possibilities to create a final connection in a more or less smart way. The program uses some heuristics to determine one feasible combination. Although this heuristics is not infinite smart, it should be easy to lead the algorithm to the desired solution, by pointing the mouse into the desired direction.

1.2.3.5. Creating A Box

Select "Box" mode from the toolbar. Choose a layer from the layer panel in which to create a new box. Left click at the first point, move the mouse to the second point and finish the box by left-clicking at the second point. Press the ESC key to cancel the operation.

Hint: A box, once created, will remain a box. For example, it is not possible to delete one vertex of it, thus forming a triangle. This is only possible for polygons.

1.2.3.6. Creating A Text Object

Select "Text" mode from the toolbar. The editor options dialog will open that additionally prompts for the text string. Don't forget to click "Apply" to take over the current string. If the dialog has been closed unintentionally, it can be reopened with the F3 shortcut.

To actually draw the text, move the mouse to the desired location and left-click to place it.

A text can be given a size which is stored in a GDS2 file (OASIS files do not provide this feature). The size of the text is only shown in the layout if a scalable text font is selected (the "Default" font is not scalable) and text scaling is enabled. In order to do so, choose a scalable font from the "Text font" selection box in the "Display" tab of the setup dialog and check the "Apply text scaling and rotation" box in the same tab.

The text can also be rotated, which is shown as well only if text scaling and rotation is enabled. To rotate a text while placing it, click the right mouse button. This will rotate the text by 90 degree counterclockwise.

1.2.3.7. Creating A Cell Instance

Select "Instance" mode from the toolbar. The editor options dialog will open that additionally prompts for some instance parameters. The most important one of course is the cell that shall be placed. Geometrically, the rotation angle can be specified, the mirror option can be set and the instance may be specified as a regular array. As an array, the instance represents multiple placements of the cell, arranged in regular grid which is specified by the two axis vectors and instance counts in each direction. Don't forget to click "Apply" to take over the current settings. If the dialog has been closed unintentionally, it can be reopened with the F3 shortcut.

To place the instance, move the mouse to the desired location and left-click to place it. While moving, the right mouse button can be used to rotate the instance by 90 degree counterclockwise. Press the ESC key to cancel the operation.

Starting with version 0.22, KLayout supports libraries. Libraries provide cells from the outside of the layout. These cells are imported into the layout and a copy is stored inside the layout. Still, KLayout maintains the reference to the original layout, so if the library changes, the cell will be replaced when the layout is loaded again. Cells imported from a library appear as "Library.Cell" in the cell tree and the layout. Here, "Library" is the library name and "Cell" is the cell name.

The library is selected from the pull-down box to the right of the cell name. You can use the "..." button to browse the cells available in the selected library.

Libraries are a convenient way to provide common layout building blocks. In addition, KLayout now also supports PCells (parametrized cells). Such cells do not have a static content, but instead they are created dynamically from a piece of code using a set of parameters specific for the PCell type. For example, a circle has two parameters: the layer where the circle should appear and the radius of the circle.

PCells are provided by libraries. If a PCell is selected from a library, the instance properties page also offers a panel to edit the PCell parameters. What parameters are available depends on the type of PCell.

KLayout comes with a standard library "Basic" which offers some basic curved shape types and a text generator.

KLayout offers a unique feature for the PCell implementation: a PCell can employ "guiding shapes". "Guiding shapes" are shapes that do not appear as layout themselves but are used by the PCell to derive its geometry from. For example the "rounded path" PCell of the "Basic" library uses a path as a guiding shape. This path is manipulated to obtain the final shape.

Guiding shapes are drawn on the cell box layer and can be manipulated with the normal shape manipulation functions (in particular move and partial edit). Also, the shape properties can be edited via the "Edit/Properties" function.

To learn more about libraries, read About Libraries. Read About PCells for details about PCells.

1.2.3.8. Moving The Selection

The whole selection can be moved in "Move" mode. If some elements are already selected, choose "Move" mode and select a reference point by left-clicking at the position. The reference point will be used as the "dragging handle" - each element is moved relative to this position. In no elements are selected when entering move mode, simply click at the element to move and place it somewhere else with a left mouse click.

While moving, the whole selection can be rotated by 90 degree counterclockwise with a right mouse button click. The ESC key will cancel the operation.

For movements, the movement direction constraint apply. See Basic Editor Options for details about the modes available. For example, in manhattan mode, only horizontal and vertical movements are allowed. The global movement constraint can be overridden by pressing Shift (orthogonal), Ctrl (diagonal) or both Shift and Ctrl (any angle) while moving the mouse.

If a move distance and direction is known numerically, "Move By" from the "Edit/Selection" menu can be used. A dialog will open that allows specification of the horizontal and vertical move distance in micrometers. Positive values move to the top or right and negative ones to the bottom or left. This dialog also applies to partial mode, so that edges or parts of a layout can be moved precisely by a certain distance in a certain direction.

In the same way, "Move To" allows one to reposition the selection to a certain point. The point which is positioned can be chosen relative to the bounding box of the selection. If first example, the lower-left corner of the selection is picked as the reference point and a certain position is given in the dialog, the selection is moved such that the lower left position of its bounding box will match the given coordinate.

1.2.3.9. Other Transformations Of The Selection

The selection can be flipped at x- or y-axis, rotated as a whole or moved by a certain distance using the functions available in the "Selection" submenu of the "Edit" menu. For example, "Flip Vertically" flips the selection at the x-axis. A selection can be rotated by an arbitrary angle using the "Rotation By Angle" function from the "Selection" submenu.

1.2.3.10. Partial Editing

When objects have to be modified after the have been created, partial editing comes into play. "Partial" refers to the fact that just parts of a polygon or path are edited. For example, just one vertex or an edge of a polygon can be moved. Partial editing mode also allows deleting single vertices or edges or to insert new ones. In partial editing mode, multiple edges or vertices can be selected, even a whole shape can be selected and can then be moved or deleted.

When moving the selected parts, the movement direction constraint applies. See Basic Editor Options for details about the modes available. For example, in manhattan mode, only horizontal and vertical movements of parts are allowed. Again, the global movement constraint can be overridden by pressing Shift (orthogonal), Ctrl (diagonal) or both Shift and Ctrl (any angle) while moving the mouse.

To enter partial mode, click on the "Partial" button in the toolbar. Parts (edges or vertices) can then be selected either by simply clicking at them or by dragging a selection rectangle. As in normal selection mode, the modifier buttons Shift and Ctrl can be used to add a selection to the existing one or to remove elements from the existing selection. Partial selection is subject to the "top level only" constraint (see Basic Editor Options for a description of the top level selection mode).

Simply clicking at an item immediately enters "move" mode. In this mode, you can position the element at the desired target location and place it there by left-clicking at the position. Press "ESC" to cancel the operation. When a complex selection is made, move mode is entered by clicking at one of the selected items (the edges or vertices, not the shape to which they belong).

When moving parts, certain constraints apply, i.e. single edges can only be moved perpendicular to their current position. In addition, the movement is confined to the editing grid.

The selected items can be deleted by using the "Delete" function from the "Edit" menu or pressing the "Delete" key. If not enough vertices remain to form a valid object, the object is deleted (i.e. a polygon with less then 3 points).

By double-clicking at an edge or path segment, an additional point is created on this edge at the cursor's position. You can create a "bend" on a path by placing two new vertices on that segment and moving the connecting segment between these vertices away from the former center line. This basically requires two double-clicks on the path's centerline, a single click on the newly formed segment and a single click to drop it at the new position.

1.2.3.11. Moving Shapes To A Different Layer

The selected shapes can be moved to a different layer as a whole. For this, choose "Change Layer" from the "Selection" submenu of the "Edit" menu. All selected shapes are moved to the layer that is the current one (marked with a rectangle) in the layer list. The shapes will not be moved across the hierarchy but just inside their cell.

All layers (source and target) must be located in the same layout. To move shapes to a different layout, use copy & paste.

1.2.3.12. Other Layer Operations

The layer specification can be edited using the "Edit Layer Specification" method from the "Layer" submenu inside the "Edit" menu. A dialog is shown in which the layer, datatype and (OASIS) name of the layer currently selected in the layer panel can be edited. On save, the shapes are then mapped to the new layer.

A layer can be cleared (either cellwise, on a cell's hierarchy or for all cells) using the "Clear Layer" method from the "Layer" submenu inside the "Edit" menu.

Layers can be copied (duplicated) using "Copy Layer" from the "Layer" submenu inside the "Edit" menu and deleted completely using "Delete Layer".

1.2.3.13. Copy And Paste Of The Selection

Of course, copy and paste is supported as usual. Shapes can be copied between layouts: by opening two layouts, shapes can be moved from one layout to another. The shapes are mapped to the same layer than they have been on in the source layout. If a layer does not exist yet in the target layout, it is created.

Shapes in the selection are simply copied to the clipboard in the way they appear in the current cell. This means, if the shapes are pasted into a different layout they are put on the same position, but flat into the current cell. This provides a way to flatten a hierarchy: choose "hierarchical selection mode" in the editor options dialog (deselect "top level only"), select the shapes to flatten and copy everything to a different cell.

In non-hierarchical selection mode ("top level only" selection mode) or by clicking on a cell frame when the hierarchy levels are limited, instances can be selected as well. When copying instances to the clipboard, two possible methods exist:

- **Shallow copy:** In this mode, just the instance is copied. When it is pasted into any target layout, the target cell of the instance is looked up and instantiated.
- **Deep copy:** Not only the instance but the instantiated cell is copied as well. When pasting that into a different layout, the target cell will be created as well. If a cell with that name already exists, a variant is created and instantiated.

1.2.3.14. Delete A Cell

To delete a whole cell, select the cell in the hierarchy browser and choose "Delete Cell" from the context menu (right mouse button). This time, three possible modes are offered:

- **Shallow delete:** Just the cell (its shapes and instances) are deleted, not any cells referenced by this cell. Since cells might no longer be referenced after that, they may appear as new top cells in the layout.
- **Deep delete:** The cell and all its subcells are deleted, unless the subcells are referenced otherwise (by cells that are not deleted). In this delete mode a complete hierarchy of cells can be removed without side effects.
- **Complete delete:** The cell and all its subcells are deleted, even if other cells would reference these subcells.

1.2.3.15. Rename A Cell

To rename a cell, select the cell in the hierarchy browser and choose "Rename Cell" from the context menu (right mouse button). You are prompted for a new name which must not exist yet.

1.2.3.16. Copy And Paste Of Cells

Whole cells can be copied to the clipboard as well. To copy a whole cell, select the cell in the hierarchy browser (make sure the focus is in that window) and choose "Copy" or "Cut" from the "Edit" menu. To paste such a cell into a target layout, choose "Paste" from the "Edit" menu.

Two copy modes are provided: deep and shallow copy. When "copy" or "cut" is chosen and the cell instantiates other cells, a dialog will been shown in which the mode can be selected:

- **Shallow copy**: In shallow mode, only the cell itself will be copied. No copies of the child cells will be created. If a cell copy is created, the cell will call the same cells than the original cell. If a cell is copied to another layout (in a different tab), the child cells will not be carried along and "ghost" cells will be created or existing cells with the same name will be used as child cells.
- **Deep copy**: In deep copy mode, the cell plus its child cells are copied. All cells will be carried along and when pasting the cell, copies of all children will be created as well.

When a cell is pasted into another layout and there is a "ghost cell" with that name, the pasted cell will replace the ghost cell. If there is a normal cell with that name, a new cell variant will be created and the name of the pasted cell will be changed by adding a suffix to create a unique name.

Copying a cell in deep copy mode from one layout to another provides a way to merge two layouts into one: simply copy the top cell of the first layout into the second one and instantiate both in a new top cell for example.

1.2.4. Advanced Editing Operations

This section covers the more advanced features of editor mode.

- Layout Transformations
- Search and Replace
- Hierarchical Operations: Flatten Instances, Make Cell From Selection, Move Up In Hierarchy
- Creating Clips
- Flatten Cells
- Resolving Arrays
- PCell Operations
- Layer Boolean Operations
- Layer Sizing
- Shapewise Boolean Operations
- Shapewise Sizing
- Object Alignment
- Corner Rounding
- Cell Origin Adjustment
- Create Cell Variants

1.2.4.1. Layout Transformations

Some functions are available to transform a whole layout. Whole-layout transformations are applied to all cells in a way that every cell will be modified in the same way. This feature is specifically useful for scaling layouts. It is worth noting that scaling a layout this way does not produce instances with magnifications.

The layout transformation functions are available in "Edit/Layout" (flip, rotate, scale, move).

1.2.4.2. Search and Replace

KLayout offers a "search and replace" function which provides a basic search and search+replace, but also a very generic and powerful extended feature called "custom queries" (see below for that). The search and replace dialog can be found in the "Edit" menu under "Search and replace".

The dialog provides four tabs in the left panel: "Find", "Delete", "Replace" and "Custom". The functionality of these four tabs is explained below.

The left side of the dialog will hold results for operations which deliver a result, for example the "Find" operation. The result is a list which displays various items, depending on the nature and parameters of the operation. If the item represents a layout object, for example a shape or a cell instance, the items selected in the list are highlighted in the layout to indicate their position. The length of the list is limited to avoid performance degradation for very long lists. The number of items shown can be configured on the configuration page.

The configuration page which allows configuration of the search and replace dialog's behavior is shown when the "Configure" button at the left bottom corner is pressed. It can be found as well in the setup dialog under "Browsers", "Search Result Browser".

Find

The functionality of the "Find" tab is simple: Various conditions can be specified and all objects matching that condition are listed when the "Find" button is pressed.

The parameters of that function involve: Object type, cell scope and object specific conditions. The object type is either "Instances" or a shape object. For shapes, the shape type can be confined to "Box", "Polygon", "Path" or "Text" which enables specific features.

The cell context can be one of:

- **Current cell**: look in the current cell and none else. Child cells are ignored.
- **Current cell and below**: look into the current cell and all child cells, where all instances of the children are considered.
- **All cells**: look into every cell individually, but don't consider the way the cells are instantiated.

Depending on the object type various parameters are available to be included in the condition. Each condition applies to one specific parameter and is usually composed of an operator (less, less than, equal ...) and a value against which the value of the parameter is checked. Length and area values are given in micron or square micron units.

String values can be matched against glob pattern using the tilde ("~") match and non-match ("!~") operators. Glob pattern are the ones used for file names on the command line and use "*" for an arbitrary sequence of characters and "?" for a single arbitrary character. Here are some examples for glob pattern:

A*	The string must start with a capital "A"
A	The string must contain a capital "A" somewhere
ABC?	"ABC" followed by any character
N{AND,OR}	"NAND" or "NOR"
A[0-9]	"A" followed by a digit
A[^0-9]	"A" followed by a non-digit

If the value field is left empty, no check is made on that parameter. All conditions which are checked must be fulfilled to make the object listed on the results page.

Delete

Similar to the find page, this function asks for an object type, a cell context and object specific conditions.

If the "Delete All" button is pressed, all selected objects are deleted.

If the "Select+Delete" button is pressed, the selected objects will first be shown in the result page, similar to the "Find" function. Then, some or all of them can be selected and deleted by pressing the "Delete" button below the list.

Replace

Similar to the find page, this function asks for an object type, a cell context and object specific conditions. In addition, the function allows specification of replacement values for the parameters. If an entry field is left empty for the replacement value, no replacement is made.

For strings with glob pattern matching, name parts can be reused in the replacement string. For example, if the operator is "~" on a text's string and the match string is "A(*)", the replacement string can be set to "B\1". "\1" means the value of the first bracket in the match string, hence this setup replaces all leading "A"'s by "B".

If the "Replace All" button is pressed, the replacement parameters are set on all selected objects.

If the "Select+Replace" button is pressed, the selected objects will first be shown in the result page, similar to the "Find" function. Then, some or all of them can be selected and the replacement is made on them the "Replace" button is pressed below the list.

Custom queries

The full power of this dialog is unleashed when using that page. Custom queries are not only able to provide the functionality of the three other pages (find, delete and replace), but provide functionality far beyond that simple scenarios.

Custom queries are statements resembling SQL statements with embedded expressions and a rich language to describe shape or cell instantiation details. A in-depth description can be found here: About Custom Layout Queries.

The custom query dialog page offers an entry field to enter the query and below an "Execute" button to run it. The most recently used queries can be pulled back into the edit field using the drop-down box below the edit field. Custom queries can be saved under a given name and reused. The list of saved queries can be manipulated with the buttons right to it. See the button's tooltips for a description of the button's functionality.

If the tab is switched to another tab and back, the custom query will be updated reflecting the query corresponding to the current functionality selected on the other tab.

1.2.4.3. Hierarchical Operations: Flatten Instances, Make Cell From Selection, Move Up In Hierarchy

Klayout provides several operations that move shapes or instances up and down in hierarchy. All these operations are accessible through the "Edit" menu in the "Selection" sub-menu.

- **Flatten instances**: Replace the selected instances by the contents of the instantiated cell. KLayout will ask, if all levels or just the first level of the cell should be expanded. If all levels are expanded, the cell will be resolved into a set of shapes in the current cell's hierarchy.

- **Move up in hierarchy**: Applies only to selections inside child cells of the current cell (thus does not make sense if 'top level only' selection mode is active). The selected shapes and instances are brought up to the current cell's level and removed from the original cell.
  A non-destructive way of moving a shape up in the hierarchy is to copy and paste the shape. This does an explicit flattening of the shapes selected when inserting them.
  Hint: the current implementation removes the selected object from its original cell. Since it only creates new copies for the selected instances, the object ist lost for all other instances of the cell. This may create undesired side effects and it is likely that this behaviour will change in future implementations.

- **Make cell from selection**: Removes the currently selected objects and places them into a new cell whose name can be specified.

1.2.4.4. Creating Clips

KLayout provides a utility to create rectangular clips from a given cell. One or more rectangles can be specified. The current cell is cut along the edges of these rectangles. For each rectangle, a new cell is created containing the clipped content for the rectangle. Finally, if more than one rectangle is specified, all the clips are combined into a master top cell which appears as a new top cell in the cell hierarchy.

The clips can be either specified by coordinates, taken from another layer (which must contain boxes which then are copied into the output as well) or taken from the rulers. In the latter case, the rulers' start and end points are taken as the corners of the clip rectangles. It is convenient therefore to create a new ruler type with a box appearance for this purpose.

Clips are done hierarchically: child cells are clipped as well, potentially creating variants (which may be shared by several clips). This way, large clips can be created from large layouts in an efficient way. **Hint:** Clipping will not work exactly if the layout contains cell instances with arbitrary rotation angles such as 45 degree.

1.2.4.5. Flatten Cells

A "flat" cell is a cell without hierarchy. This means that the cell contains only shapes, but no instances of child cells. Flat cells are disconnected from other cells, hence flattening is a way to "freeze" the contents of a cell: when a cell is flat, changing any other cell does not have an effect on this cell or in other places of the cell. On the other hand, flat cells store each shape individually, hence cannot make use of data compression be reuse of geometry.

A hierarchical cell can be flattened by choosing "Edit/Cell/Flatten Cell" or "Flatten Cell" from the cell list context menu. The flatten operation offers some options, i.e. the number of hierarchy levels to flatten and how to deal with child cells which become obsolete through this operation ("orphan cells"). By enabling this "prune" option, all child cells are removed when they are no longer needed. Otherwise, new top level cells will appear in that case - these are the cells which are not longer instantiated.

"Flatten" can also be applied to instances. In that case, the cell instance is removed and replaced by the objects inside this cell. So instance flattening is a way to pull the contents of a cell into the parent cell. One reason for doing so is to make the cell contents accessible for editing, without having to change the child cell itself. This prevents potential side effects when editing a cell would make the edits visible in other places.

Instance-wise flattening is available by choosing "Edit/Selection/Flatten Instances". Again, options are available to choose the number of hierarchy levels to flatten and how to treat orphan cells.

1.2.4.6. Resolving Arrays

Instance arrays are handy to produce large regular arrangements of cells. Sometimes it is necessary to resolve these arrays - for example to remove or modify a single instance from the array. One way to achieve that is to create cell variants (see Create Cell Variants, but this feature will also create a copy of the instantiated cell.

The "Resolve Arrays" function available in "Edit/Selection" allows resolving of an array into individual instances which then can be edited individually.

1.2.4.7. PCell Operations

PCells can be created from shapes if the PCell is derived from a "guiding shape". Specifically the Basic library PCells support derivation from guiding shapes with a few exceptions. So for example, a round-cornered polygon can be created from a normal polygon by selecting the polygons, choosing "Edit/Selection/Convert To PCell" and selecting the "Basic.ROUND_POLYGON" for the PCell.

PCells can be converted to normal cells by choosing "Edit/Cell/Convert Cell To Static" or "Edit/Layout/Convert All Cells To Static". Normal (static) cells can be edited individually but do no longer offer parameters to control the look of the cell.

1.2.4.8. Layer Boolean Operations

KLayout now comes with a set of boolean operations. These operations are available in the "Layers" submenu of the "Edit" menu ("Boolean Operations" and "Merge" functions). A dialog will open that allows specification of mode, input layer(s), output layer and certain other options.

- **AND**: intersection. The output layer will contain all areas where shapes from layer A and layer B overlap.
- **A NOT B**: difference. The output layer will contain all areas where shapes from layer A are not overlapping with shapes from layer B.
- **B NOT A**: difference. The output layer will contain all areas where shapes from layer B are not overlapping with shapes from layer A.
- **XOR**: symmetric difference. The output layer will contain all areas where shapes from layer A are not overlapping with shapes from layer B and vice versa.

In addition, a **MERGE** operation is provided, which is a single-layer operation that joins (merges) all shapes on the layer. As a special feature, this operation allows selecting a minimum overlap count: 0 means that output is produced when at least one shape is present. 1 means that two shapes have to overlap to produce an output and so on. This does not apply for single polygons: self-overlaps of polygons are not detected in this mode.

All operations can be performed in three hierarchical modes:

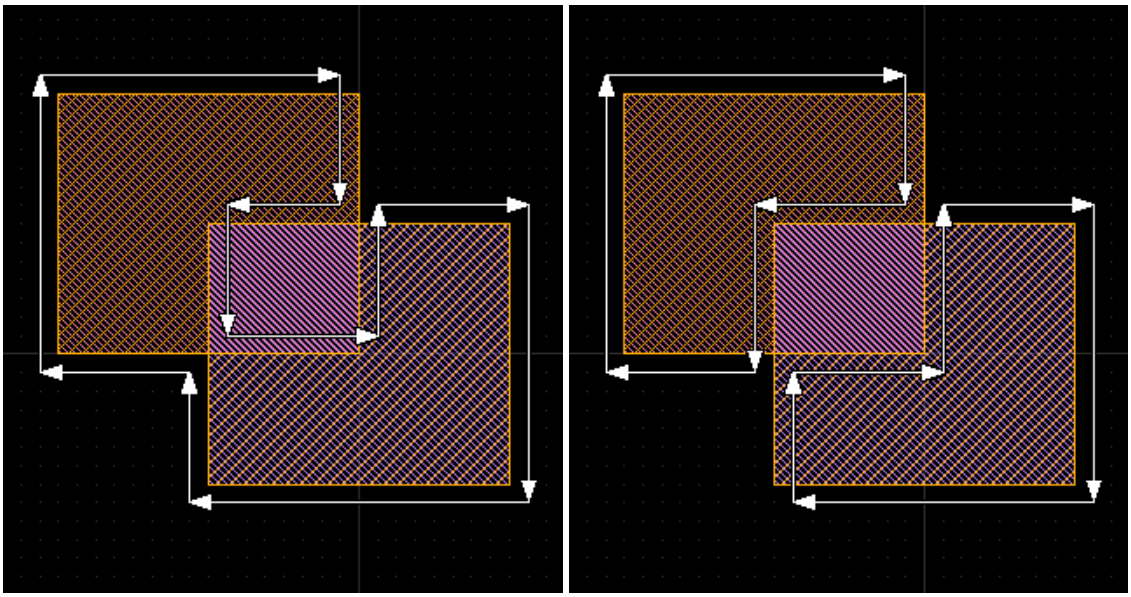
- **Flat**: Both layers are flattened and the results are put into the current top cell.
- **Top cell**: perform the operation on shapes in the top cell only.
- **Cellwise**: perform the operation on shapes of all cells below the current top cell individually. This mode is allowed only if the layouts of both inputs and output are the same.

For the first two modes, the source and target layout can be different, provided that all layouts are loaded into the same view. This allows combining layers of different layouts. For example to compare them using a XOR function.

As a special feature, KLayout's boolean implementation allows choosing how "kissing corner" situations are resolved. KLayout allows two modes:

- **Maximum coherence**: the output will contain as few, coherent polygons as possible. These polygons may contain points multiple times, since the contour may return to the same point without closing the contour.
- **Minimum coherence**: the output will contain as much, potentially touching polygons as possible.

The following screenshots illustrate the maximum coherence (left) and minimum coherence (right) modes for a XOR operation between two rectangles.



The boolean operations are currently implemented flat and based on a full-level edge representation. This means, that the complete layer is flattened (if "flat" mode is requested) and converted into a set of edges which the processor runs on. This will lead to huge resource requirements for very large layouts and is not recommended for such applications currently.

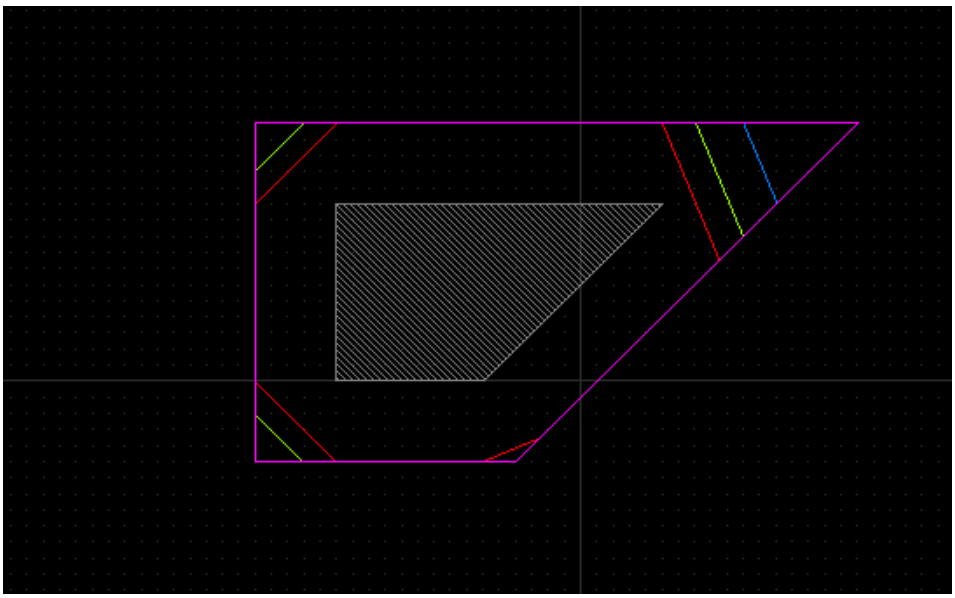
The boolean processor is based on an iterative approach to cover grid snap effects which makes it highly accurate but somewhat slower than a single-pass scanline implementation. Performance penalty is about 2x (two times slower) compared to an efficiently implemented single-pass algorithm.

1.2.4.9. Layer Sizing

A sizing operation allows growing or shrinking of the shapes of a layer by a given offset, which is applied per edge. Positive values will enlarge the shapes while negative values will shrink the shapes. The offset can be given separately for horizontal and vertical direction. However, the sign of both values must be identical (i.e. "0.5,0" or "1.0,0.2", but not "0.2,-0.2").

The sizing function can be found in the "Layers" submenu of the "Edit" menu. A dialog will open that allows specification of input and output layers, sizing value in micron: a single value for same sizing in x and y direction or comma-separated list of two values (i.e. "0.2,0.1").

As for the boolean operations, hierarchical mode and kissing corner resolution can be specified (see Layer Boolean Operations for a description of these modes). In addition, the cutoff strategy for sharp edges can be chosen from strict to virtually unlimited. The following screenshot demonstrates the effect for "strict" (red) to "weak" (purple) cutoff modes.



1.2.4.10. Shapewise Boolean Operations

Boolean operations are also available on selected shape sets. These operations use the concept of "primary" and "secondary" selection. The primary selection contains all shapes that are selected in the first step. The secondary selection contains all shapes that are selected in additional steps using the "Shift" modifier key.

The following operations are available in the "Selection" submenu of the "Edit" menu:

- **Merge**: merge all shapes in the primary and secondary selection and write the results to the layer of the primary selection.
- **Intersection**: Compute the intersection (AND) of primary and secondary selection and write the results to the layer of the primary selection.
- **Subtraction**: Compute the difference (A NOT B) of primary (A) and secondary (B) selection and write the results to the layer of the primary selection.

1.2.4.11. Shapewise Sizing

The selected shapes can be sized with a given enlargement and shrink distance, similar to the layer operation but with less options. The sizing function can be found in the "Selection" submenu of the "Edit" menu. A dialog will open that prompts for the sizing value (one value for same sizing in x and y direction in micron or two comma-separated values for different sizing in x and y direction).

1.2.4.12. Object Alignment

This operation use the concept of "primary" and "secondary" selection. The primary selection contains all shapes that are selected in the first step. The secondary selection contains all shapes that are selected in additional steps using the "Shift" modifier key.

The object alignment function allows aligning of all objects in the secondary selection to the objects in the primary selection (i.e. objects in the primary selection define the reference points but are not moved). An "object" can be a shape or an instance of a cell. Cell instances are referred to by their bounding box which can be either computed from the visible layers alone or from all layers.

Alignment can be specified differently in horizontal and vertical direction. Horizontal alignment can be "none" (no change), "left" (align left sides), "center" (align centers) or "right" (align right sides). Vertical alignment can be "none", "bottom", "center" or "top".

The alignment function can be found in the "Selection" submenu of the "Edit" menu. A dialog will open which allows specification of the alignment mode and bounding box computation mode for cell instances.

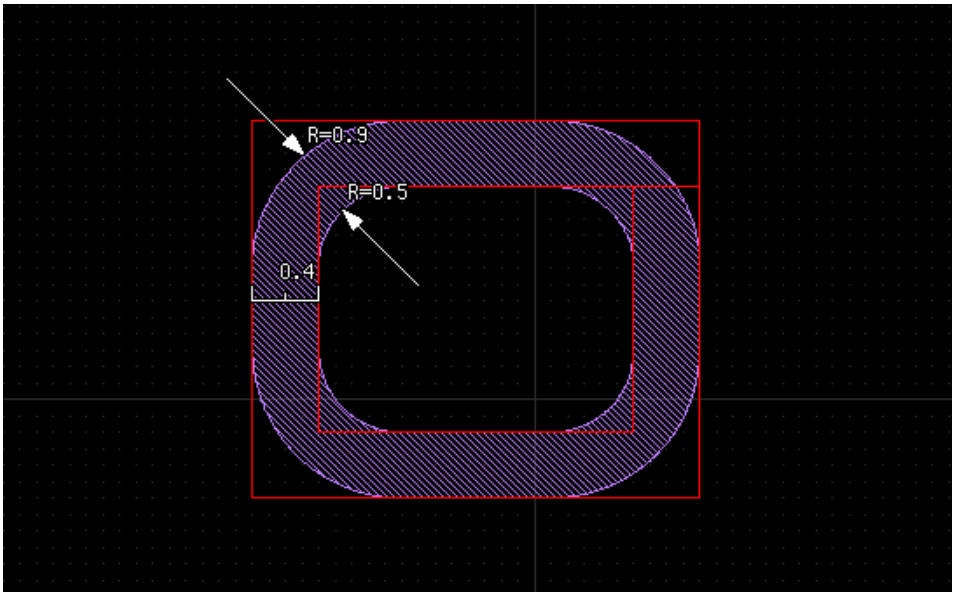
1.2.4.13. Corner Rounding

In some applications, i.e. power devices, it is desirable to have round corners instead of sharp corners to limit the electrical field. Klayout now offers a convenient way to create such structures. The basic idea is to draw the structures with sharp, 90 degree corners and then "soften" the corners by rounding them to a given radius. The resulting polygons can then be written to GDS files, even though GDS does not have the concept of "soft" (or circular) geometries.

The interesting part is: **the corner rounding function can be re-applied on such geometries on a polygon basic**. That means, that even if such a modified polygons are saved to GDS or otherwise modified, the original geometry can be reconstructed and the corner radius can be changed. No special geometrical objects or special GDS annotation is required to achieve this. This requirement imposes some (probably weak) limitations:

- The number of points per corner must not be too small (currently at least 32 on the full circle)
- The original geometry must not exhibit sharp corners and the original segments must be at least twice the corner radius in length.
- The corner segments must be perceivable as such, i.e the angle between adjacent edges must be "nearly" 180 degree. This imposes some restrictions on the minimum length of such a segment and on the accuracy by which they can be expressed in database units. This boils down to a certain length limit in terms of database units.

The following screenshot illustrates the round corners function. As can be seen in this example, it is necessary to allow a different radius specification for "inner" and "outer" corners.



The corner rounding function operates on selected shapes. It can be found in the "Selection" submenu of the "Edit" menu. A dialog will open which allows specification of the radius values and the desired resolution. If the selected polygon already has rounded corners, the corner rounding will be removed and the original polygon reconstructed before the new corner rounding is applied. By specifying "0" for the radius, the original sharp corners will be recovered.

1.2.4.14. Cell Origin Adjustment

The cell origin is important for a cell because this point is the instantiation anchor for cell instances. The cell origin adjustment function allows shifting the origin to a certain place relative to a cell's bounding box. This can be either the center, a corner or the middle of an edge of the bounding box. The bounding box can either be computed from all or just from the visible layers.

If the "Adjust instances in parents" check box is clicked, the instances of the cell will be adjusted in the opposite direction. Hence, the cell effectively does not change, but locally the origin of the layout will be shifted.

The cell origin adjustment function can be found in the "Cell" submenu of the "Edit" menu.

1.2.4.15. Create Cell Variants

KLayout offers a feature that is very useful when you want to edit a single instance of a cell rather the cell itself. Editing a cell means that the changes applied to the cell appear at all places where this cell is placed. That may not be desired - if you want to modify a layout in one particular place you may not want to have side effects at other places.

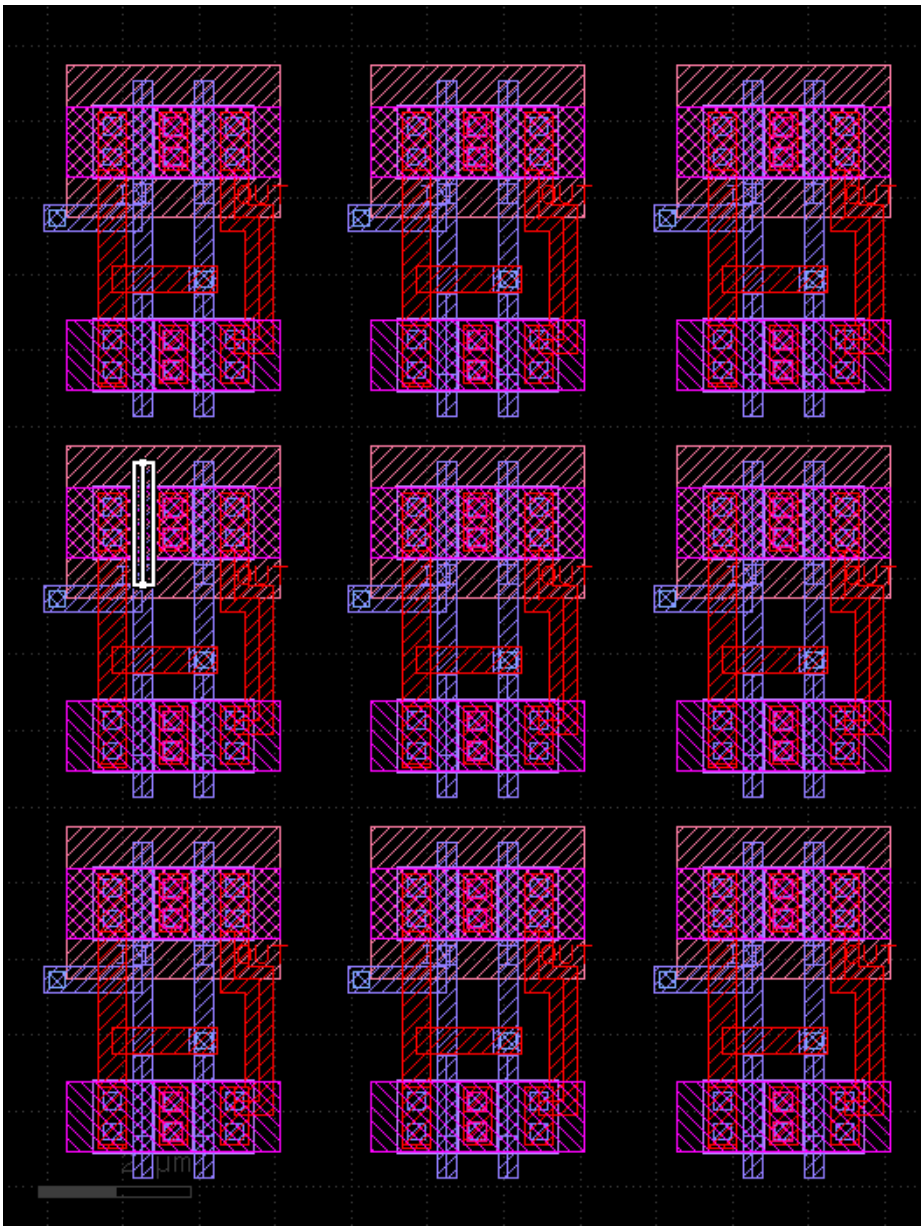
The usual way of achieving that is to copy a cell, replace a particular instance by the new cell and edit the new cell. This can be tedious, in particular if there is an array instance where just one instance must be modified. In that case, the array has to be split so the single instance is isolated before it can be replaced with a new cell.

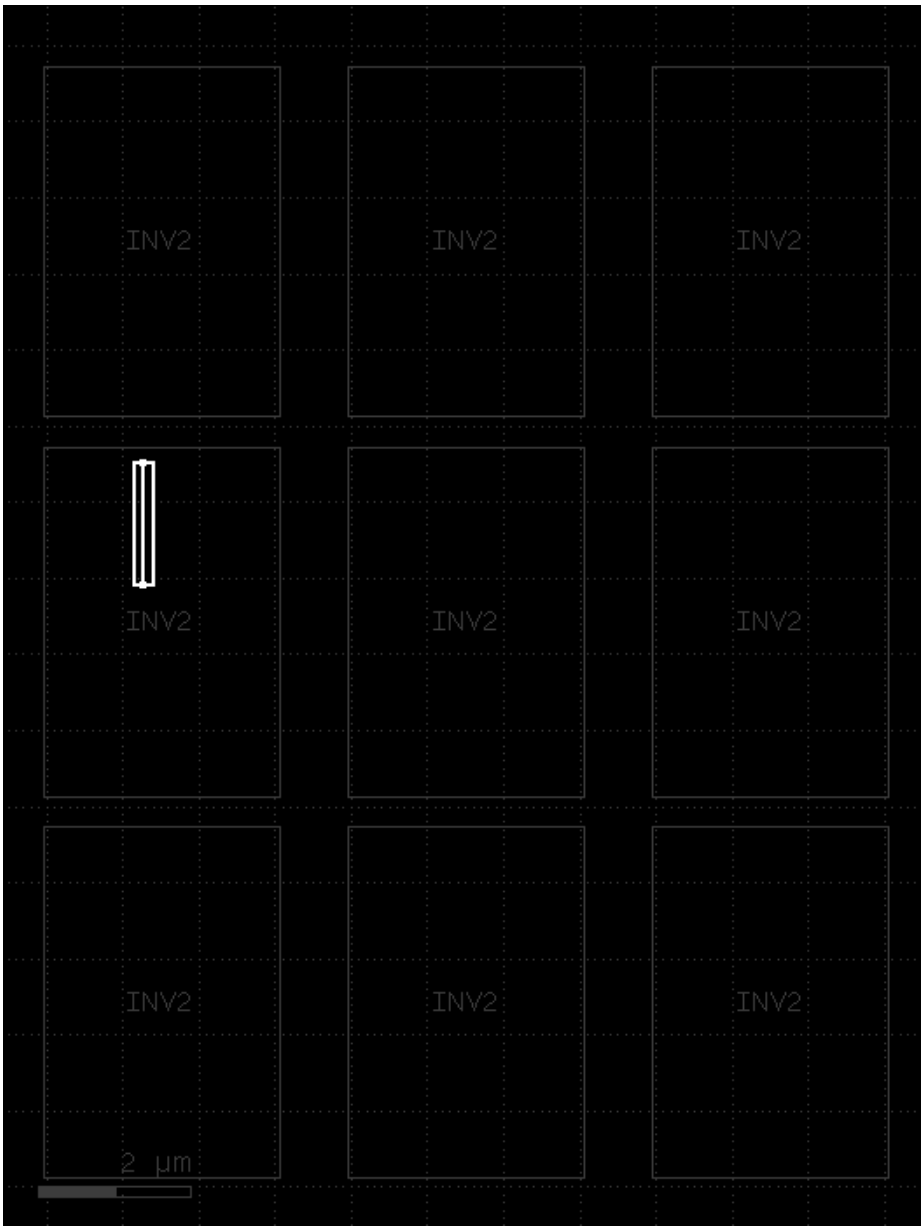
KLayout offers a feature that automates this task. It is found in the "Edit/Selection/Make Cell Variants" menu. It basically works like this: For all selected objects it will follow the hierarchy up to the current cell. It will create new cell copies for all cells found along that path and use these new cells instead of the original ones.

The effect is, that after using "Make Cell Variants", the selection can be modified (i.e. deleted) without having any undesired side effects.

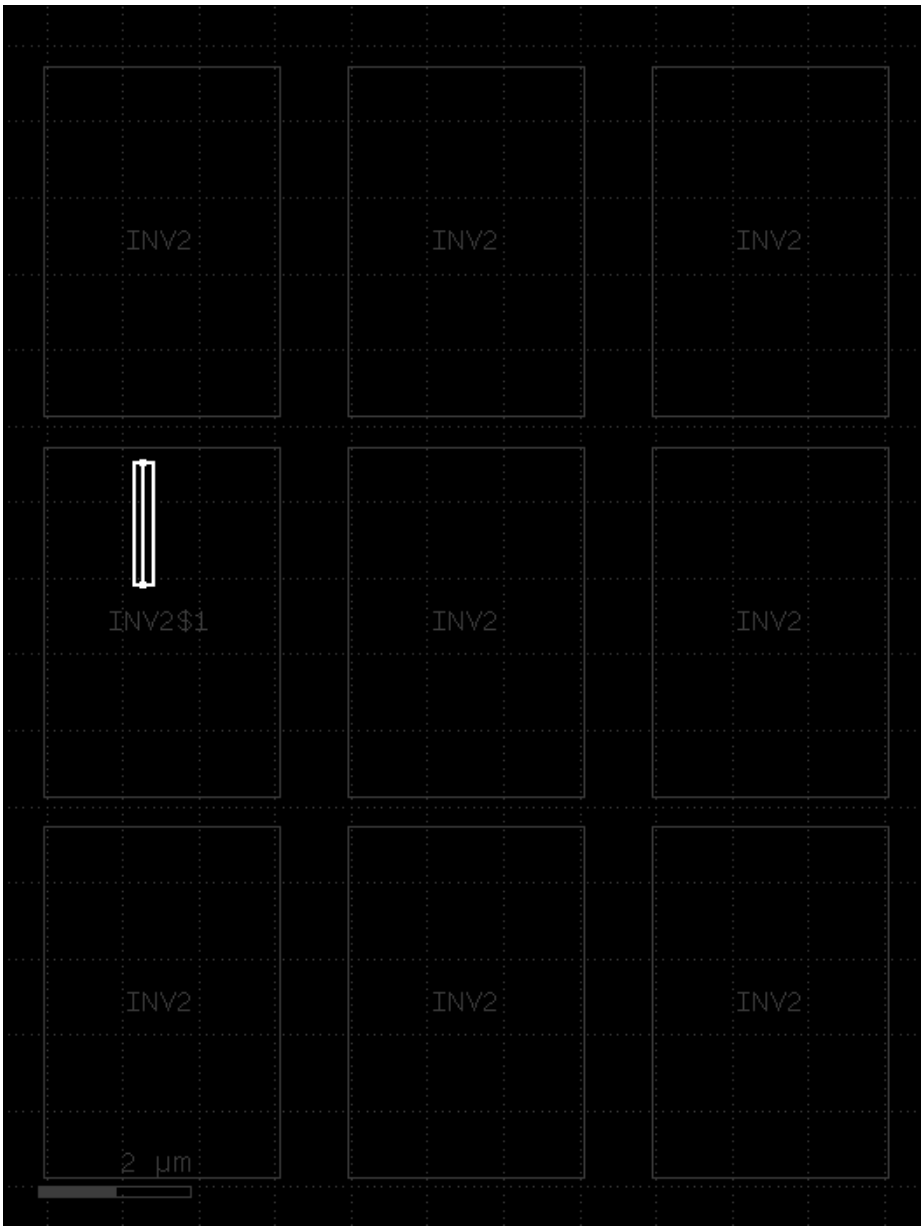
This feature can also transform array instances and isolate certain instances of the array. The following screenshots demonstrate that feature.

This is the initial situation: a cell is instantiated 9 times in a 3x3 array and one shape inside one of these instances is selected. The two following screenshots show the full-level hierarchy view and just the top level hierarchy to demonstrate that the cell is placed 9 times.

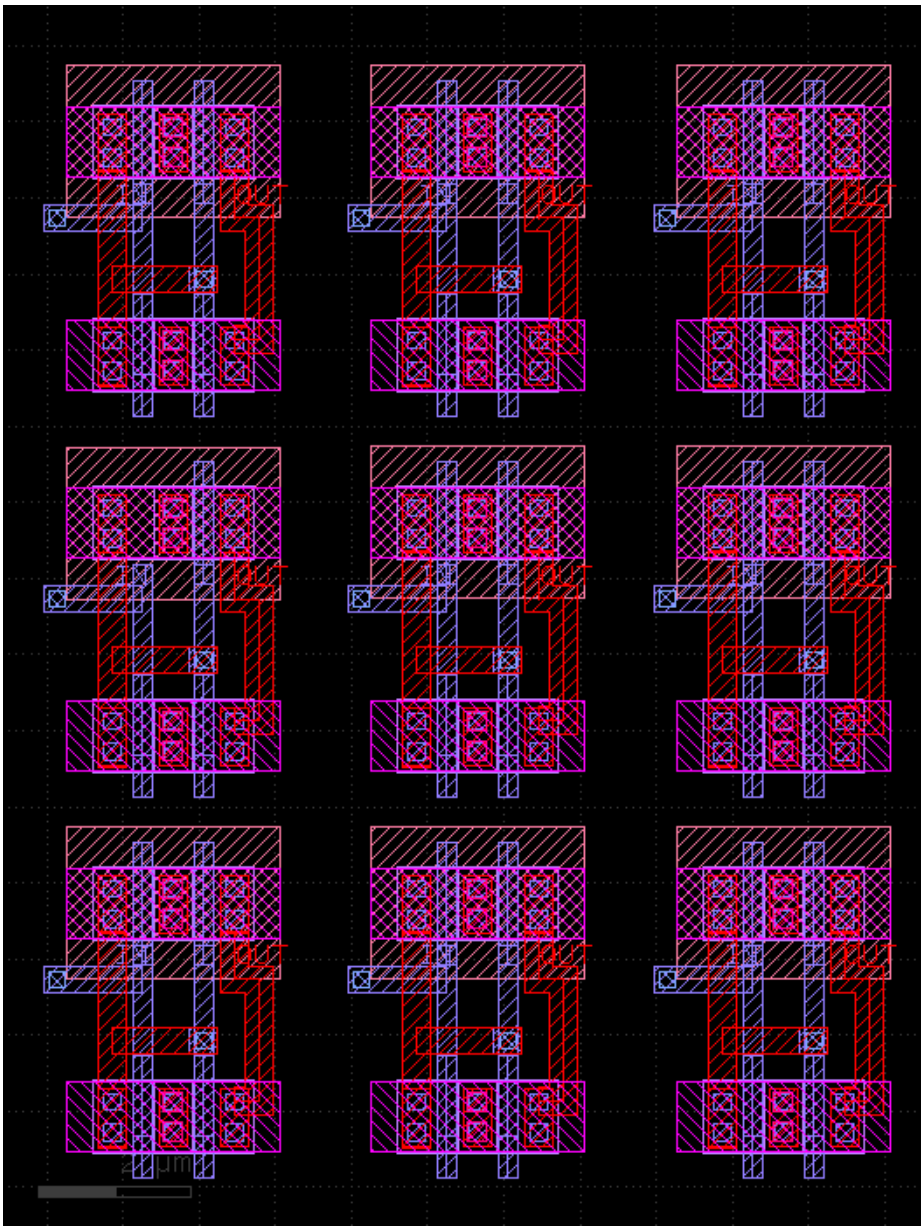




If that shape would be deleted, it would disappear in all instances. Imagine that is not intended. You can now use the "Make Cell Variants" feature to create a cell variant for the one instance that we have selected the shape in. After we did so, one instance is replaced by a copy of the cell which is called "INV2$1". The picture looks like this:



We can now delete the single shape without any side effect on the other instances:



1.3. Advanced Topics

This section describes some more advanced features of KLayout:

1.3.1. The XOR Tool

The XOR tool performs a geometrical XOR (also ANOTB and BNOTA for asymmetric differences) on two layouts by performing the respective boolean operations layer by layer. The XOR tool is started using "XOR Tool" from the "Tools" menu). Currently, the tool compares all or just the visible layers. Currently, it compares layers from one layout vs. the identical layers from the other layout.

The current implementation employs a flat XOR processor. This limits the application somewhat to small and medium sized layouts and does not make use of hierarchy, which basically excludes applications for very hierarchical layouts (i.e. memory arrays). The memory footprint associated with the flat approach can be mitigated by using the tiling feature which performs the operation on a tile with limited size. This does not reduce the run times but the memory requirements.

The XOR tool allows specification of tolerances. Basically a tolerance is an undersizing step following the boolean operation. This way, small markers can be suppressed. This is particular useful to remove markers resulting from tiny differences between the layouts being compared. Multiple tolerances can be specified. In that case, multiple undersize steps are performed to create sets of layers with different tolerances each. For example, a tolerance specification of "0,0.001,0.005,0.010" will create four sets (marker categories) containing all difference markers and others for markers indicating differences larger than 1nm, 5nm and 10nm.

Tiling can be enabled by entering a tile size into the entry box. For semi-flat layouts such as standard cell blocks, a tile size of 1000 micron is a good starting point. The choice of the tile size mainly determines memory requirements.

The XOR tool allows sending the output either to a marker database or to another or one of the input layouts. The mode can be selected with the "Output" drop-down box. If output is sent to one of the original inputs, it is mandatory to specify a layer offset which maps the original layer to a new layer. An offset of "1000/0" for example means, that differences between shapes on layer "16/0" will be sent to "1016/0" for the first tolerance category and "2016/0" for the second.

Finally, the XOR can be confined to a region. This saves time if differences in parts of the layout are of interest. To select a region use the drop-down box next to "From region":

- **All:** Do an XOR on the whole area (default)
- **Visible region:** Confine the XOR to the visible part of the layouts
- **Clipped to ruler:** Draw one or many rulers to specify the region to which the XOR shall be confined. Each ruler specifies a region given be the extension of the ruler. If you use a box-type ruler, the ruler gives a better visualization of the region

1.3.2. The Diff Tool

As the XOR tool, the Diff tool performs a comparison of two layouts. In contrast to the XOR tool, it does a cell-by-cell and object-by-object comparison and reports differing cells, instances and geometrical objects. In effect, the comparison is more strict and not purely geometry-related. It does not verify the identity of the layouts on mask level but rather the exact identity of the objects that comprise the layout file. On the other hand, the Diff tool usually detects the actual changes rather than their effect on geometry.

Usually, that kind of comparison is very sensitive to "cosmetic" changes, i.e. cell renaming. KLayout's Diff tool tries to mitigate this effect with these features:

- Before it does the cell-by-cell comparison it tries to detect cells which have been renamed by comparing their instantiation. That way, it can compare the right cells even though their names may be different. The basis of that functionality is a cell matching algorithm. This algorithm compares cells by taking into account their bounding boxes, shape counts per layer, number of instances and other parameters. The algorithm will choose a partner cell which matches closest with respect to these parameters. If that scheme fails, it is possible to revert to name matching by unchecking the option "Don't use names to match cells".
- It allows some level of control over the strictness of the compare. For example, cell arrays can be expanded before the individual instances are compared. By default, some second-order information like users properties or certain text properties is not compared.
- The diff tool can also work in "XOR" more. In that mode, the differences found are used to provide input for a subsequent, polygon-only XOR step. The result is a fair approximation of a true, as-if-flat XOR which delivers a superset of the true XOR's results. It may report some locations as being different which if fact are not, but it will not fail to report differences where there are some. Compared with the XOR tool's functionality, some options are missing (i.e. tolerance), but the performance is much better.

The Diff tool is found in the "Tools" menu. In this dialog:

- Select layout A and B in the "Input" section.
- Uncheck "Don't use names to match cells" to revert to pure name matching. Cells which have been renamed will not be compared against then.
- Check "Run XOR on differences" to select the "XOR mode".
- Check "Summarize missing layers" to have missing layers reported as one difference instead of one per shape.
- Check "Detailed information" to receive detailed information about every difference. Without that option, only the number of differing shapes or instances is reported.
- Check "Expand cell arrays" to compare individual instances of array instances.
- Check "Exact compare" to include second-order information (i.e. user properties, text orientation) in the compare.

The Diff tool will create a marker database and show the results in the marker database browser.

1.3.3. The Fill (Tiling) Utility

The fill utility creates a regular pattern of fill unit cell instances in certain areas of a layout. This feature is usually referred to as "tiling" or "fill". It is based on a rectangular unit cell which is repeated in x- and y-direction to fill the available space. In most cases, the intention is to fill empty areas in the layout to enhance the layout uniformity for a better process performance.

Before the fill utility can be used, a fill cell must be prepared in the layout that is filled. The dimension of the cell are defined by a box drawn on an arbitrary layer. This box must represent the "footprint" of the cell. This is the space that one instance will cover in the region to be filled.

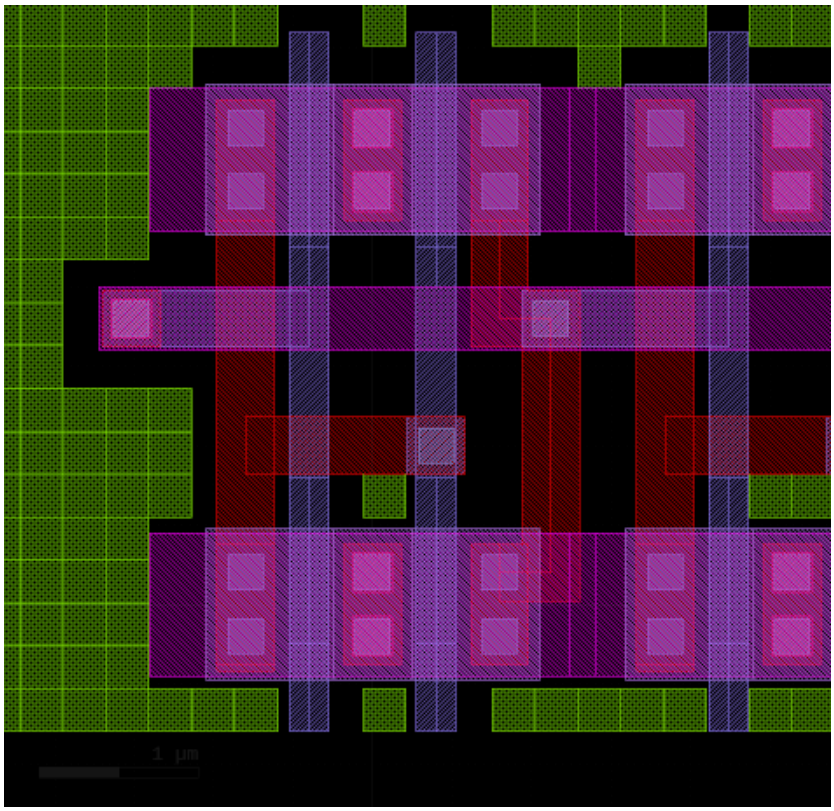
The fill utility is found in the "Utilities" sub-menu of the "edit" menu and is available in edit mode only. To use this utility, open the dialog and

- Select the outer boundary of the fill region ("what to fill"). Available choices are: Full cell, the interior or the polygons on a given layer, the interior of all selected polygons, a single box or an area defined by a ruler.
- Specify if the fill area should keep a certain minimum distance to the border of the fill region.
- Specify the regions within the fill region which must not be filled. Available choices are: All layers (don't create fill over any polygon drawn), all visible layers (don't create fill over any polygon visible), all selected layers or don't exclude anything.
- If the fill tiles must keep a certain minimum distance from the exclude regions, specify that distance in the "Spacing around exclude areas" entry field.
- Specify the fill cell and the boundary layer which defines the cell's footprint in the "Fill Cell" group.

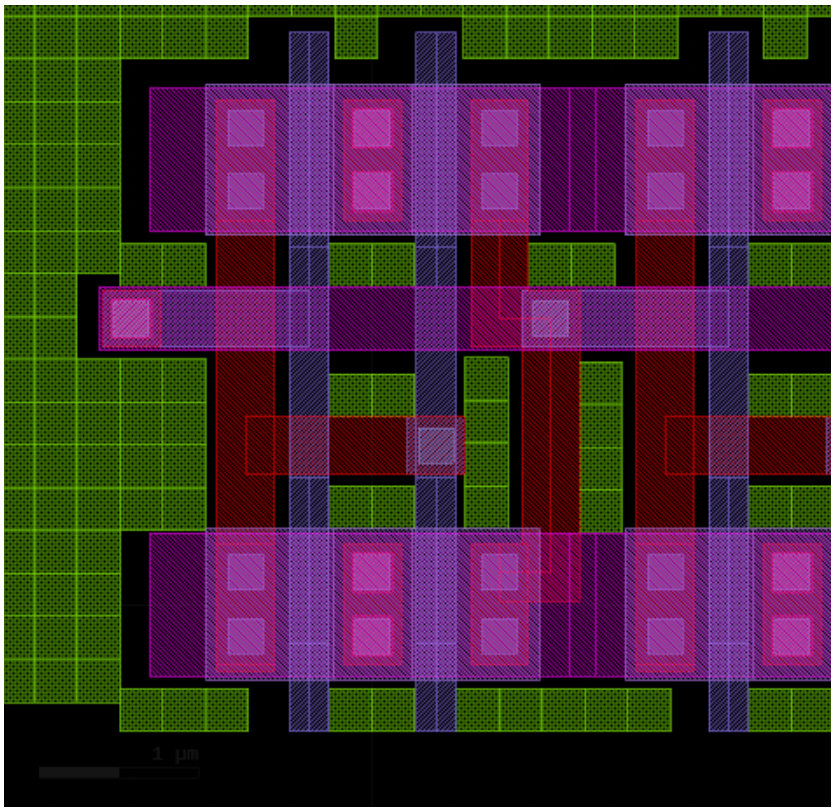
By default, the fill utility operates on a fixed raster. This can lead to a poor fill efficiency in some cases. The fill utility offers a "Enhanced fill" option, where it tries to find a cell arrangement which is not necessarily on a common raster but provides a better fill performance. In addition, second-order fill is supported. In that case, a second - usually smaller - fill cell can be specified which is used to fill the remaining areas of the layout.

The following screenshots show the effect of the different fill modes for some artificial fill problem.

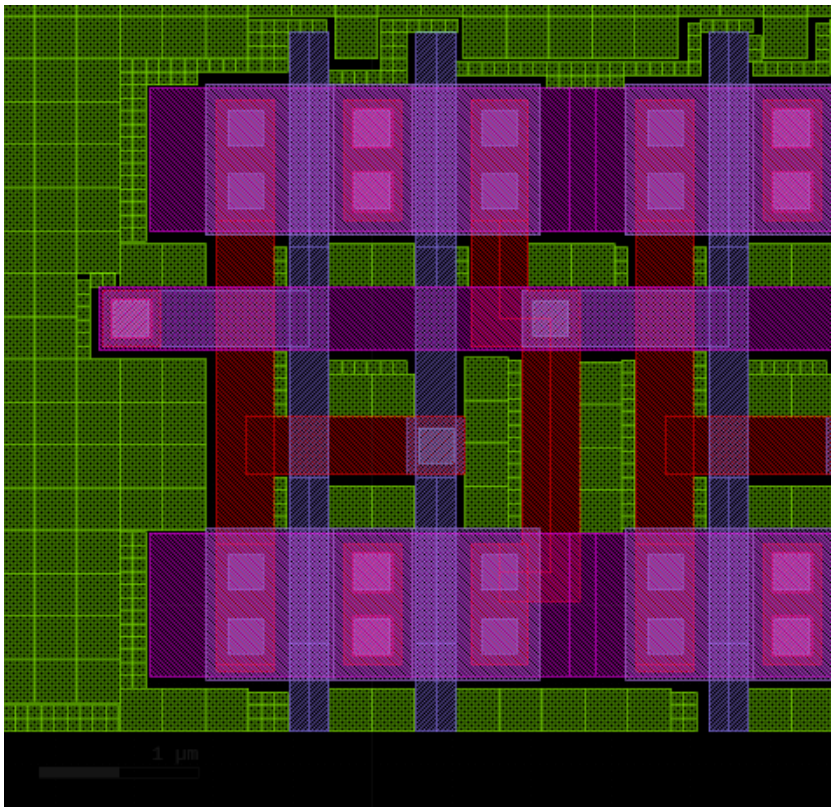
Default:



Enhanced:



Enhanced plus second order:



1.3.4. Import Gerber PCB Files

Gerber PCB import allows creating GDS layout data from Gerber PCB files or to add Gerber files to GDS files as new layers. The import function supports a majority of the RS274X features for artwork files and a couple of different formats for the drill files. The importer will take a set of files and convert them to layout geometry. The importer offers some functionality to adjust the data appropriately, i.e. to define output layers and apply geometrical transformations. Another basic capability is to merge the geometry of a layer to remove overlaps and join paths into larger polygons.

Because of the manifold options, the import specification can become pretty complex. Therefore, it can be saved into a file (suggested suffix is ".pcb") in XML format which contains the importer specifications. Once such a file is created, KLayout can read this file like usual stream files (i.e. it can be specified on the command line) and use it as a recipe to import the associated Gerber files.

The PCB import function is available in the "File" menu ("Import/Gerber PCB"). Different entry points are provided that start a new project, open an existing project or to continue with the last project.

The basic workflow to import PCB data is:

- Specify the directory where the PCB data files are located (the "base" directory).
- Specify the import mode (the destination of the layout data).
- Decide about the layer mapping mode: free layer mapping or metal stack mapping. Free layer mapping allows an arbitrary mapping between PCB layers and GDS layers. This specification is the most flexible one but is tedious to enter. Metal stack mapping is easier to specify but confined to mapping a set of PCB files to a metal/via/metal stack scheme.
- Specify the files, GDS layers and PCB to GDS layer mapping.
- Specify a transformation if desired, either by specifying mapping points or a transformation directly.
- Decide about further options (i.e. merging, database unit, top cell name etc.).

The basic decision is how to specify the layer mapping. In free mode, the specification requires these steps:

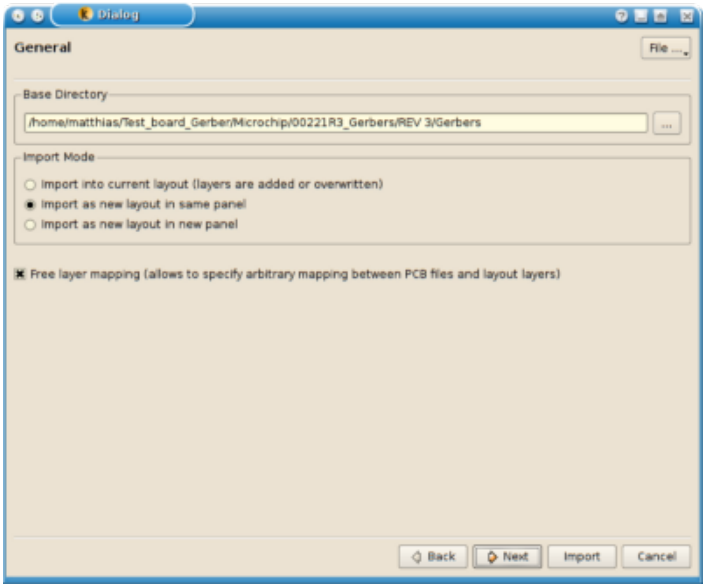
- Specify the files to load (in the dialog on the "Files" page).
- Specify a list of output layers.
- Fill the input to output mapping matrix which assigns one or many output layers to each input file.

In layer stack mode, the specification workflow consists of these steps:

- Specify the GDS layer stack (the complete stack available for mapping PCB data into). The idea is basically to put another set of series of metal/via/metal layers on top of the GDS layer stack. The PCB layer closest to the die surface is placed into the first metal layer which is supposed to be the first above the on-chip layers.
- When the GDS layers are set, specify how many metal and drill hole files the PCB file set contains and whether the chip will be mounted on the top or bottom of the PCB. The latter decides in which order the PCB layers are assigned to GDS layers (remember, the first GDS layer will be the PCB layer closest to the die surface).
- Enter file names for the artwork files corresponding to metal layers.
- Specify file names for the drill files and what metal layers are connected by the (plated) drill holes. Since a drill hole can connect multiple layers in the stack, a connection information is always of the type "from metal, to metal" with the drill holes connecting all metal layers between "from" and "to".

The import dialog

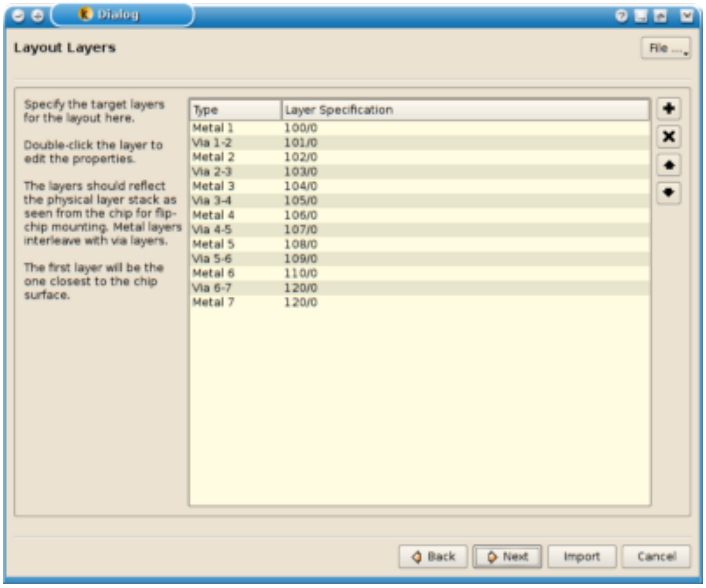
The import dialog is organized in multiple pages that reflect the workflow for the import specification. On every page, the "File" menu button allows saving the current settings as a PCB import project, to open an existing project or to create a new project and to restart from scratch.



The first page offers some basic options:

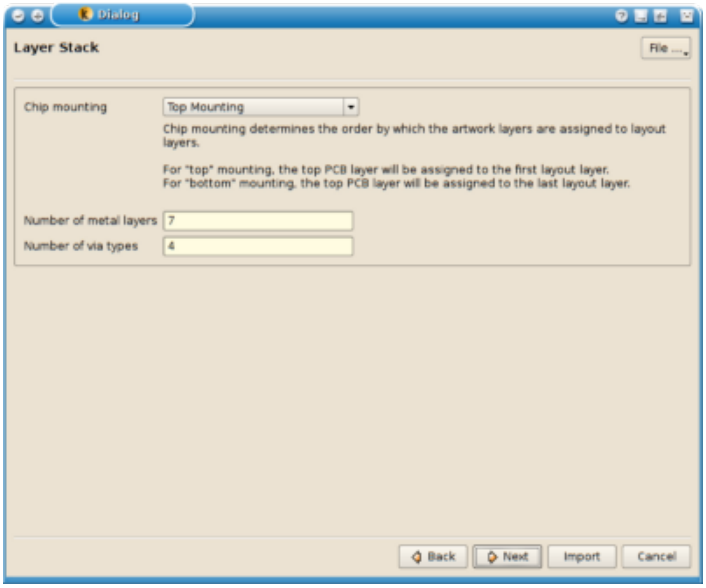
- **Base directory:** this is the directory where all the PCB files are found. Not necessarily all files must be located there but are looked for relative to this directory. If all files are moved, just the base directory must change. The base directory is not stored in a project file. Instead, the base directory is the directory where the project file is stored. Basically this implies, that all data files will be referred to relative to the project file.
- **Import mode:** PCB data can be imported into the current layout (into the current cell). Usually, it this case, layers will be added to the current layout. Alternatively, a layout can be created which will be either placed into a new panel or added to the current panel.
- **Layer mapping mode:** Specify here whether to use free or layer stack mode. Check the box to use free layer mapping mode.

The layer stack flow

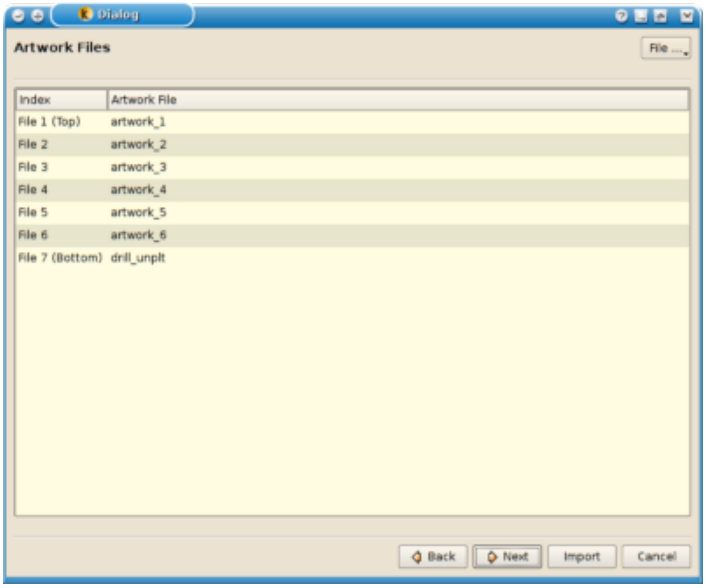


In the layer stack flow, on the first page, a sequence of metal and via layers must be specified. The assignment of metal and via layers is done automatically. The sequence is always a metal layer followed by a via layer. The number of layers must be odd so the last layer is a metal layer again. Via layers will connect the adjacent metal layers only.

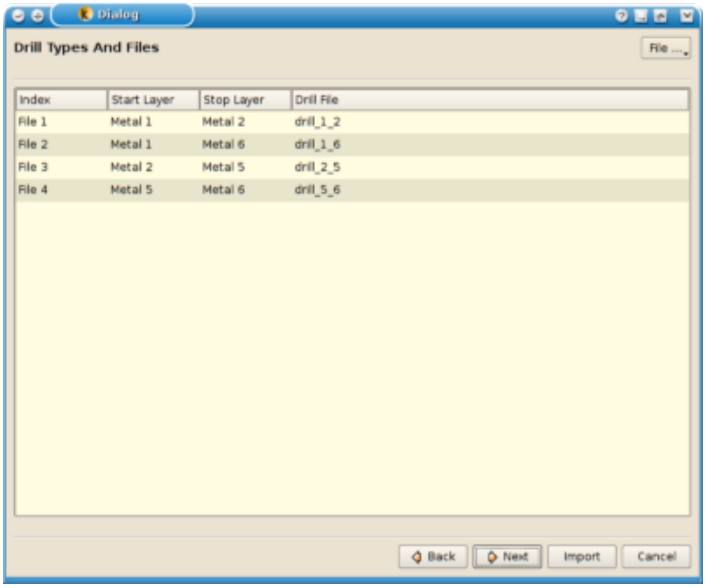
Use the "+" button to add new layers. Move layers by selecting them and moving them up or down with the arrow buttons. Use the "X" button to remove all selected layers.



On the next page, the number of artwork and drill files needs to be specified. Later, the actual files need to be entered and assigned to metal or via layers. In addition the chip mounting position needs to be specified. In "top mounting" mode, it is assumed that the chip is placed surface down on the top (first) PCB layer. Thus the first metal above the chip stack will be the top PCB layer. In "bottom mounting" mode, the last PCB metal layer will be the first metal layer above the chip stack.

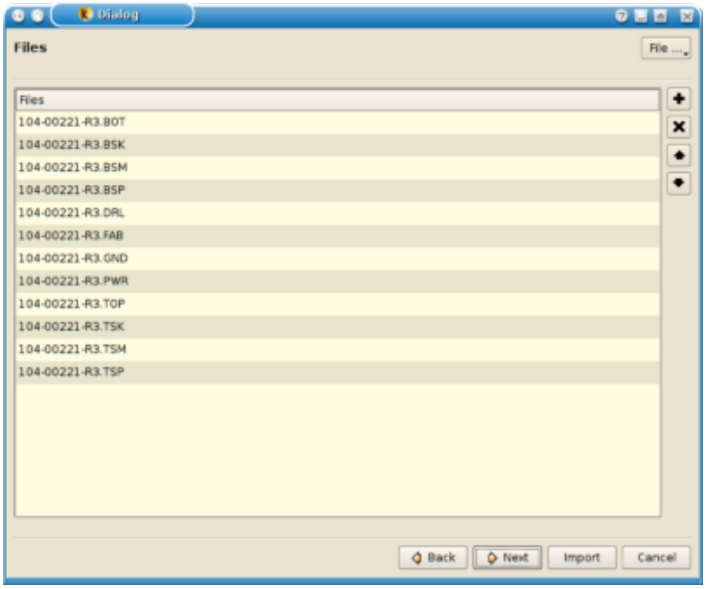


On the "Artwork Files" page, the artfile file names must be entered. They are automatically assigned to the respective metal layers. The assignment order depends on the mounting mode.

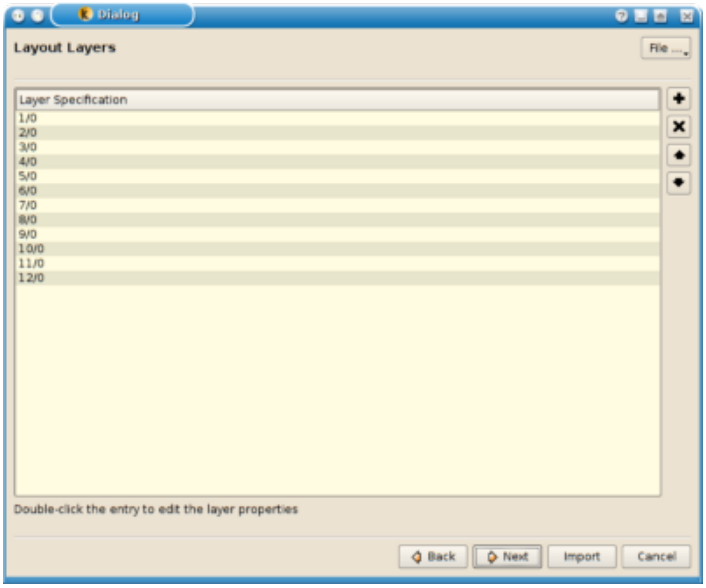


On the "Drill Files" page, the drill file names must be entered. Each drill file describes a certain drill step, which can connect multiple metal layers. On this page, this specification must be made. The first and last metal layer connected by the plated hole must be specified. The corresponding via layers will then be used to create via shapes.

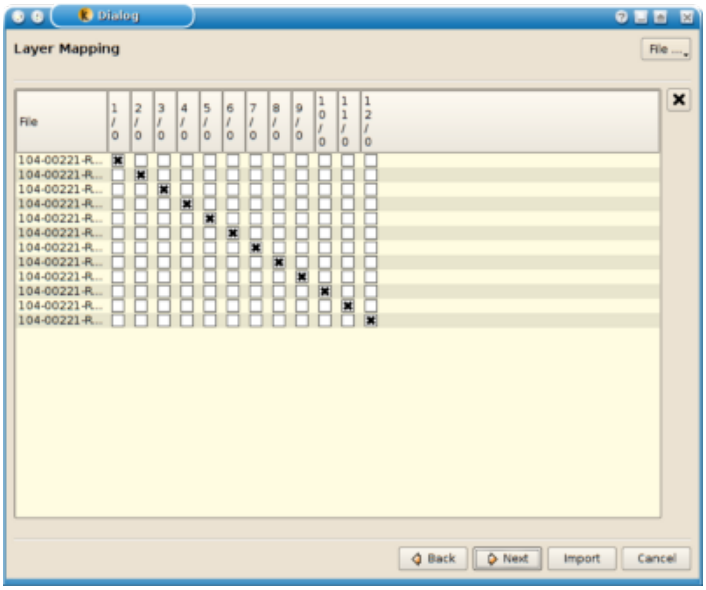
The free layer mapping flow



On the "Files" page, all PCB data files must be specified. This includes artwork and drill files. The order is not important but it is recommended to follow the physical stacking. This simplifies the assignment to GDS layers later. Use the arrow buttons to move the selected entries up or down. Use the "X" button to delete files from the list and use the "+" button to add new files.

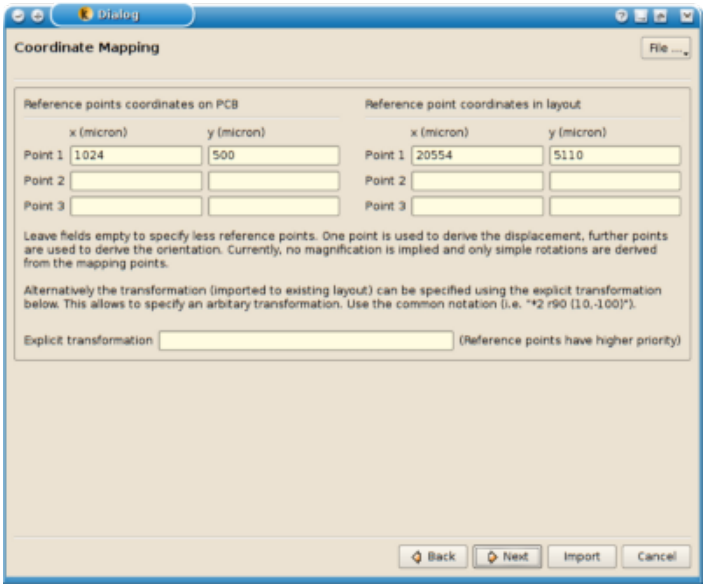


On the "Layout Layers" page all target layers must be specified. Provide a list with all layers that are used as target layers for the import. Again, the order is not important but maintaining a technological order will simplify the assignment in the next step. As on the previous page use the arrow buttons to move selected entries and the "+" and "X" button to add new entries and deleted the selected ones.



On the "Layer Mapping" page, each file can be assigned to one or may GDS layers. The assignment is described in form of a matrix where an "x" means that the file or layer given by the row is imported into the layer given by the column. A file can be imported into multiple layers which basically will duplicate the shapes. Click at the boxes to set or reset the mark. Use the "X" button on the left to reset all marks for the rows selected.

General options



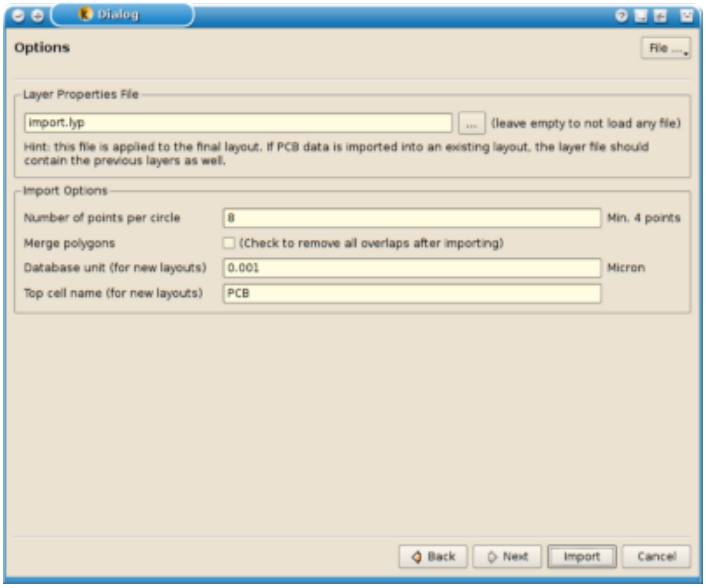
The "Coordinate Mapping" page allows specification of the transformation of the PCB data into the GDS space. Since PCB and GDS rarely share the origin, a transformation can be specified which is applied to the shapes when importing them.

A specification can be made in two ways:

- **By specifying matching points:** The transformation will be computed such that the given PCB coordinates are mapped to the given GDS coordinates. Up to three coordinate pairs can be given. If one coordinate pair is given, a displacement is derived. If two coordinate pairs are given, the rotation is computed as well (only multiples of 90 degree are supported currently). If three coordinate pairs are give, the algorithm can derive mirroring as well.

- **By explicitly specifying the transformation:** The transformation can be specified explicitly in the lower entry field. The format is "x,y" for a simple translation (x, y are given in micron units), "rx" or "mx" for a rotation by the angle "x" or mirroring at the line with angle "x" and "*x" for a magnification of "x". All specifications can be combined, i.e. "r90 170,-5100" specifies a rotation by 90 degree and displacement by 170 micron in horizontal and -5.1 mm in vertical direction.
  For a comprehensive description of that string, see Transformations in KLayout.

Hint: Both specifications can be combined, i.e. one coordinate pair can be given to define the displacement and the rotation can be specified explicitly.



Finally, on the "Options" page, various options can be set:

- **Layer properties file:** If specified, this layer properties file will be loaded after the layers have been imported. The file is specified relative to the base directory.
- **Number of points per circle:** KLayout resolves the circular apertures commonly used in PCB layout into polygons to perform geometrical operations. This options allows choosing how many points will be used for the approximation of a full circle. Less points will mean less accurate representation but smaller polygons hence better performance on boolean operations used to compute clear areas for example.
- **Merge polygons:** If this option is set, all polygons will be joined if they overlap or touch. Note, that merging also happens implicitly if clear layers are used because the boolean operations used to cut out clear regions will implicitly merge the previous layout. This implicit merging cannot be disabled.
- **Database unit and top cell name:** This option allows choosing the database unit and top cell name for new layouts. This applies only, if the import mode implies a new layout.

1.3.5. Import Other Layout Files

This function can merge other layouts into the layout loaded. Merging means that the hierarchy of the specified layout is inserted into the given layout. Different modes are available that control the way how the hierarchy is merged. This function is available in the "File" menu as "Import/Other File Into Current".

The workflow for importing a different layout is this:

- Specify the file to input. At least the file name is required. Additionally, a cell can be specified. In that case, only the cells referred to by the given cell (directly or indirectly) are imported. Reader options can be specified separately for the import. Reader options are applied the same way than the reader options are used for the standard load function.
- Specify the import mode. The modes are described below.
- Specify the layer mapping. Either the shapes are imported on their original layer or an offset can be used that will be added to the layer to form the target layer of the import. An offset of "1000/0" for example specifies to add 1000 to the layer and use the original datatype.
- Specify an optional transformation. The imported layout will be transformed accordingly. The transformation can be specified explicitly or with up to three points which are mapped onto each other.

Four import modes are available that control how the hierarchy of the imported layout is inserted into the existing layout:

- **Merge**: in this mode, the contents of the imported cell will be put into the current cell and the child hierarchy is added below the current cell.
- **Extra**: in this mode, new top level cells containing the hierarchy tree of the imported cell or cells will be created. In this mode, multiple cells can be imported if the imported layout contains multiple top cells. Leave the cell specification empty for this.
- **Instantiate**: the imported cell will be instantiated into the current cell as a separate hierarchy.
- **Merge hierarchy**: The fourth mode is a little bit more complex. Basically it works like "Merge", but identifies corresponding cells and merges the contents for the corresponding imported cells into the original cells. The algorithm identifies corresponding cells by requiring that the flat instances of the imported child cell exactly equal the flat instances of the corresponding original cell (where flat refers to the instances of a cell in the context of the current cell). This is done by selectively thinning out the candidate list and finally employing a name similarity measure to resolve ambiguities.

The import function will create new cell names using the "$x" suffix to avoid name ambiguities.

1.3.6. The Net Tracing Feature

The net tracing function allows tracing of a net by detecting touching shapes that together form a conductive region. It features specification of a stack of metal (or in general "conductive") layers optionally connected through via shapes. The net tracing algorithm will follow connections over the via shapes to form connections to other metal layers. The material footprint can be derived from single layout layers or a boolean combination of several layers. For example, this allows selecting source and drain regions of transistors by subtracting the poly region from the active area region.

Single net tracing

This algorithm is intended for extracting single nets and employs an incremental extraction approach. Therefore extraction of a single small net is comparatively fast while extraction of large nets such as power nets is considerably slower compared to hierarchical LVS tools currently.

The net tracing function can be found in the "Tools" menu. The user interface allows tracing of multiple nets which are stored in a list of nets extracted. If labels are found on the nets, these are used to derive a net name. Beside that, the cells which are traversed in the net extraction are listed, so the cells being connected by this net can be identified.

Before nets can be extracted, a layer stack must be specified. Press "Layer Stack" on the user interface to open the layer stack dialog or use "Edit Layer Stack" in the "Tools" menu. Layers must be specified in the "layer/datatype" notation. The via specification is optional. If no via layer is specified, both metal layer shapes are required to touch in order to form a connection. If a via layer is specified, a via shape must be present to form the connection.

The layer stack (also referred to as "Connectivity") is also a technology component. This means if you select a technology for layout, the technology specific stack is selected. To edit the stack for all technologies, use the Technology Manager ("Tools/Manage Technologies") and use the "Connectivity" section from the technologies.

A connectivity definition can specify multiple stacks, out of which one has to be selected. This is useful, if your technology setup features multiple routing metal schemes for example. You do not need to setup different technologies for each scheme. If you have declared multiple stacks, you can choose one in the "Trace Net" dialog in the drop-down button right of the "Layer Stack" and the "Configuration" button at the bottom left.

Within the stack definition, KLayout allows specification of symbolic layers and to use boolean expressions. That way it is possible to assign meaningful names to layers (i.e. "POLY" or "VIA1") and to use derived layers (i.e. "ACTIVE-POLY" for the source and drain regions of a CMOS transistor). Read more about these features in Connectivity and Symbolic Connectivity Layers.

Once a layer stack has been defined and selected, a net can be traced by pressing the "Trace Net" button and clicking on a point in the layout. Starting from shapes found under this point, the net is extracted and listed in the net list on the left side of the net tracing dialog. If "Lock" is checked, another net can be traced by clicking at another point without having to press the "Trace Net" button again.

Sometimes you encounter large nets (e.g. power nets). When you click on such a net, the tracer will start running and the extraction will take a long time. If you're not interested in the details of such nets, you can limit the depth of the net tracing. This means, after a specified number of shapes is encountered, the tracer will stop and report the shapes collected so far as an incomplete net.

To configure the depth, enter the desired number of shapes in the "Trace depth" box at the bottom of the trace dialog. NOTE: the actual number of shapes in the net may be a litte less than the specified depth due to internal marker shapes which are taken into account, but are not delivered as parts of the net.

The "Trace Path" function works similar but allows specification of two points and let the algorithm find the shortest connection (in terms of shape count, not geometrical length) between those points. If the points are not connected, a message is given which indicates that no path leads from one point to the other.

The display of the nets can be configured in many ways. The configuration dialog is opened when "Configure" is pressed in the trace net dialog. Beside the color and style of the markers used to display the net it can be specified if and how the window is changed to fit the net.

Tracing all nets

This algorithm is borrowed from the LVS feature, where the scenario is extended by device recognition and netlist formation. In the context of the net tracer, nets consist of the connected shapes but don't attach to devices. As LVS extracts all nets in one sweep, using this feature in the net tracer will deliver all nets at once. Although this is a richer information output, the tracing of all nets is typically faster then tracing a single, big net such as power nets. The LVS net extractor also supports hierarchical processing which gives a considerable performance improvement and more compact net representations.

To extract all nets, use "Trace All Nets" from the "Tools" menu. It will start extracting the nets immediately. It will take the connectivity definition from the standard, single-net net tracer. You can edit this layer stack either from the single net tracer UI, from the technology manager or "Edit Layer Stack" from the "Tools" menu.

After the net tracer has finished, the netlist browser dialog opens. In this dialog you can:

- Browser the circuit hierarchy (taken from the cell hierarchy) in the left half of the central view.
- Browse the nets of the circuits in the right half of the view. Clicking on a net will highlight the net.
- Configure the net highlighting behavior. Use the "Configure" button.
- Export all or selected nets to layout, save the netlist (with shapes) to a file, load it back from a file and manage the netlist database. Use the "File" menu button in the right upper corner.
- Search for net names (if labeled) and circuits using the search edit box.
- Navigate through the history using the "back" and "forward" buttons at the top left.

Extracted nets are written and read in a KLayout-specific format called "L2N" ("layout to netlist"). This format contains both the nets and the shapes that make up a net. This way, the traced nets can be saved and retrieved later.

1.4. Design Rule Check (DRC)

The DRC feature of KLayout is described here. The "Basics" section describes the basic concepts and the "Runsets" section the DRC language.

- Design Rule Checks (DRC) Basics
- DRC Runsets

Further information about the DRC features can be found here: DRC Reference.

1.4.1. Design Rule Checks (DRC) Basics

KLayout supports design rule checks beginning with version 0.23. The capabilities of the DRC feature include:

- Basic DRC checks such as checks for minimum width and space.
- Layer-generation methods such as boolean operations and sizing.
- Extended geometrical checks such as overlap, enclosure and inside and outside checks.
- Support for edge objects derived from polygons or as output from other functions. Edge objects are useful to implement edge-related operations, for example selective sizing.
- The capability the work with multiple input layouts.
- Support for text object layers. Text objects are convenient for tagging polygons or labelling nets.
- Cell filtering, local (region-constrained) operation.
- A tiling approach for large layouts which can be configured to make use of multiple CPU cores.
- A hierarchical option.

The DRC functionality is controlled by a DRC script. A DRC script is basically a piece of code which is executed in the context of the DRC engine. The script language is based on KLayout's integrated Ruby interpreter and wraps the underlying object model into a lean and expressive language. Script execution is immediate. That means, that it is possible to embed conditional statements or loops based on the result of a previous operation. It is also possible to code low-level operations on shapes inside the script, although this will be considerably slower than using the functions provided.

Output of the DRC script can be sent to a layout layer or a report database. The report database is visualized in the marker database browser.

Basic scripts

Runset writing is described in detail in DRC Runsets. Here is a simple example for a DRC script:

```
report("A simple script")

active = input(1, 0)
poly = input(6, 0)
gate = active & poly
gate.width(0.3).output("gate_width", "Gate width violations")

metal1 = input(17, 0)
metal1.width(0.7).output("m1_width", "M1 width violations")
metal1.space(0.4).output("m1_space", "M1 spacing violations")
```

This script will compute the gate poly shapes from the active and poly layers using a boolean "AND". The active layer has GDS layer 1, while the poly layer has GDS layer 6. On this computed gate layer, the script will perform a width check for 0.3 micrometer. In addition a width and space check is performed on the first metal layer, which is on GDS layer 17.

Let's take the script apart:

- `report("A simple script")`
  This line will instruct the script to send output to a report database. The report database is shown in the marker database browser. The description text of the report database is given in brackets.
- `active = input(1, 0)`
  This line will create an input layer. "Layers" are basically collections of shapes, edges or edge pairs (edge pairs are objects created as output of check methods). "input" is a function which delivers a layer object. Checks and other functions are performed on those

layer objects in the spirit of object-oriented programming and the underlying Ruby interpreter. "active" will be a variable that holds such an object.
The parameters of the "input" method specify where to take the input from, in that case GDS layer 1 and database 0. In that simple form, the first layout loaded into the current view is used for input.

- `poly = input(6, 0)`
  This line will create another input layer for poly silicon (from GDS layer 6, datatype 0).

- `gate = active & poly`
  This line will compute the boolean "AND" of active and poly layers. The "&" is the operator for the boolean "AND" operation which computes the intersection of active and poly. The result will be sent to a new layer and a new layer object is created and put into the "gate" variable. This layer object is a temporary one and will not appear in the output but can be used in subsequent operations.

- `gate.width(0.3).output("gate_width", "Gate width violations")`
  This line combines two operations into one statement: first is performs a width check against a minimum width of 0.3 micrometer using the width method on the "gate" layer. A "method" is a function performed on a specific object and the notation used in the DRC script is the ".". "gate.width(0.3)" will perform a width check on the gate layer and create a new layer object with "edge pairs" for each violation. "Edge pairs" are marker objects consisting of two edges or partial edges which describe where two edges violate the check condition. In the simple geometrical checks, there are always two edges involved in such a violation - hence such a violation is best described by a pair of edges.
  The result of the check is sent to a report database category using the "output" method. Again this is a method called on an object, in that case the edge pair collection returned by the width check. The parameters of the output method describe a formal name and a readable description of the category.

- `metal1 = input(17, 0)`
  As before, this statement will fetch input from GDS layer 17 and datatype 0 and create a layer object representing that input.

- `metal1.width(0.7).output("m1_width", "M1 width violations")`
  As for the gate, this statement will perform a width check (this time for 0.7 micrometer) and output the violation markers to a report database category.

- `metal1.space(0.4).output("m1_space", "M1 spacing violations")`
  And again a geometrical check: this time a space check for minimum space of 0.4 micrometer.

The script can be written in several alternative forms. Here for example is a very brief version of the gate width check:

```
(input(1, 0)*input(6, 0)).width(0.3).output("gate_width", "Gate width violations")
```

Some aliases are provided for the boolean operation, so if you prefer object-oriented notation, you can use the "and" method:

```
input(1, 0).and(input(6, 0)).width(0.3).output("gate_width", "Gate width violations")
```

or the functional notation:

```
and(input(1, 0), input(6, 0)).width(0.3).output("gate_width", "Gate width violations")
```

Dimensions can be given in different ways:

```
# floating-point (will default to micrometer)
gate.width(0.3).output("gate_width", "Gate width violations")

# floating-point with unit
gate.width(0.3.micron).output("gate_width", "Gate width violations")
gate.width(300.nm).output("gate_width", "Gate width violations")

# integer values will give dimensions in database units!
```

```
gate.width(300).output("gate_width", "Gate width violations")

# as variable
min_width = 300.nm
gate.width(min_width).output("gate_width", "Gate width violations")
```

Installing and running scripts

To create a DRC script, choose "Tools/DRC/New DRC Script". KLayout will open the Macro development IDE and create a new script. The first thing to do is to give the script a proper name. The cursor is already on the name - just enter a new name and press "Enter".

The next step is to give the DRC script some description. Press the "Properties" button in the macro editor's toolbar and enter a description text - this is the text that will appear in the menu entry.

The DRC script will now appear in "Tools/DRC" with the description and selecting that entry will run the script.

To edit the DRC script, choose "Edit DRC Scripts" from "Tools/DRC" or enter the macro editor IDE and select the DRC category (the tab above the macro list).

DRC scripts can be shared or installed globally like normal macros. They can be put into the "macros" folders, but the preferred way is to deploy them in a folder called "drc" beside the "macros" folder. KLayout will scan various places for DRC scripts or macros, including the installation path and the application folder (usually in the home directory). Both places can be used to store DRC scripts.

For the more experienced user, DRC scripts are basically just macros bound to a DRC interpreter instead of the plain Ruby interpreter. You can create and run DRC scripts like ordinary macros. Use "Macros/Macro Development" to enter the macro IDE and create a DRC scripts with the "add macro" function (the "+" button). Choose "DRC scripts" as the template to use (in the "General" section). DRC scripts are by default bound to the DRC menu, but that can be changed in the same way than for any ordinary macro.

See About Macro Development for more details about the macro development facility.

By default, DRC scripts are put into the DRC category of the macro IDE. Macro categories are a way to organize macros, but do not imply a certain runtime behavior. Hence, DRC scripts can be put into any other location beside the "drc" folders selected by the DRC category.

DRC scripts can be executed in the debugger like ordinary macros and breakpoints or single-stepping can be used to debug DRC scripts. Behind the scene, the DRC commands are mapped to Ruby functions, so stepping into such commands will reveal the code behind the DRC functions.

DRC scripts are stored in ".lydrc" files in KLayout's macro format. Those are XML files containing the script code in the text element. KLayout also recognizes plain text files with the extension ".drc", but those files are usually lacking the necessary meta-information that allow KLayout to bind them to a menu entry. Hence such files can only be run from the macro IDE.

Using KLayout as a standalone DRC engine

KLayout can be used as a non-interactive DRC engine using a specific kind of DRC scripts. Since there is no "current layout" in standalong engine mode, those DRC scripts have to specify input explicitly using the "source" function. The same way, output has to be specified explicitly using either "target" to create an output layout or "report" to create an output report database (see DRC Reference: Global Functions for details about these functions).

Here is an example. It reads layer 1, datatype 0 from "input.gds", sizes it by 200 nanometers and writes the output to "out.gds", layer 10, datatype 0:

```
source("input.gds")
target("out.gds")
input(1, 0).sized(200.nm).output(10, 0)
```

Here is another example which saves the results to category "sized" of the report database file "out.lyrdb":

```
source("input.gds")
report("out.lyrdb")
input(1, 0).sized(200.nm).output("sized")
```

To run these DRC scripts, save the scripts to a file with suffix ".drc" and run it like shown below (replace "my.drc" by your file). It is recommended to disable some of the features not used in that case and put KLayout into non-interactive batch mode with "-b":

```
klayout -b -r my.drc
```

"-b" will disable all of the user interface functionality and puts KLayout in engine mode in which no display connection is required on Unix. Implicit loading of macros from the various search locations is disabled (you can still load some with "-rm") and the configuration file is neither read nor written, which causes less I/O and avoids write conflicts between different instances of KLayout.

1.4.2. DRC Runsets

This document will give a detailed introduction into the writing of DRC runsets. See also DRC Reference for a full reference of the DRC functions.

Runset basics

Runsets are basically Ruby scripts running in the context of a DRC runset interpreter. On that level, DRC runsets work with very few classes, specifically:

- Layers ("DRC::DRCLayer" class): layers represent input from the original layout or are created by functions generating information. Layers can be used as input for other methods or methods can be called in layers.
- Sources ("DRC::DRCSource" class): sources represent layout objects from where input is taken from. One default source is always provided - the default layout from where the data is taken from. More layout sources can be created to specify input from other layouts. Sources also carry information how to filter the input - for example cell filters or the working region (a rectangular region from which the input is taken).

Some functions are provided on global level and can be used without any object.

The basic elements of runsets are input and output specifications. Input is specified through "input" method calls. "input" will create a layer object that contains the shapes of specified layer. The results are output by calling the "output" method on a layer object with a specification where the output shall be sent to.

In general, the runset language is rich in alternatives - often there are more than one way to achieve the same result.

The script is executed in immediate mode. That is, each function will immediately be executed and the results of the operations can be used in conditional expressions and loops. Specifically it is possible to query whether a layer is empty and abort a loop or skip some block in that case.

Being Ruby scripts running in KLayout's scripting engine environment, runsets can make use of KLayout's full database access layer. It is possible to manipulate geometrical data on a per-shape basis. For that purpose, methods are provided to interface between the database access layer ("RBA::..." objects) and the DRC objects ("DRC::..." objects). Typically however it is faster and easier to work with the DRC objects and methods.

Including other files

The DRC script language is based on Ruby which delivers many native language features. Basically, inside a script you can include another script through "load". This will read a file and execute the content of this file in the context of the script it is loaded into.

Unfortunately, "load" creates a local context for variables. Hence it's not possible for example to use "load" to read a file that defines variables for further use in the DRC script.

To overcome this problem, KLayout offers a specific extension which embeds another file into the source by employing some kind of preprocessing. This way, a file can be included into another one like it was pasted at this place.

The notation is this:

```
# %include to_include.drc
```

which will include "include.drc". If no absolute path is given, this file is looked up relative to the file it is included in.

The file name can be put in quotes as well. Expression interpolation is supported (for the notation see About Expressions). Hence it is possible to access environment variables for example like this:

```
# %include $(env("HOME"))/to_include.drc
```

Because Ruby does not see the original files, some internals (e.g. introspection) will report wrong file names and line numbers. In most cases - for example when using "__FILE__" or "__LINE__" or when receiving stack traces and errors - the file names and line numbers will correctly refer to the source files before include file processing.

Input and output

Input is specified with the "input" method or global function. "input" is basically a method of a source object. There is always one source object which is the first layout loaded into the current view. Using "input" without and source object is calling that method on that default source object. As source is basically a collection of multiple layers and "input" will select one of them.

"input" will create a layer object representing the shapes of the specified input layer. There are multiple ways to specify the layer from which the input is taken. One of them is by GDS layer and datatype specification:

```
# GDS layer 17, datatype 0
l = input(17)

# GDS layer 17, datatype 10
l = input(17, 10)

# By expression (here: GDS layers 1-10, datatype 0 plus layer 21, datatype 10)
# All shapes are combined into one layer
l = input("1-10/0", "21/10")
```

Input can be obtained from other layouts than the default one. To do so, create a source object using the "layout" global function:

```
# layer 17 from second layout loaded
l = layout("@2").input(17)

# layer 100, datatype 1 and 2 from "other_layout.gds"
other_layout = layout("other_layout.gds")
l1 = other_layout.input(100, 1)
l2 = other_layout.input(100, 2)
```

Output is by default sent to the default layout - the first one loaded into the current view. The output specification includes the layer and datatype or the layer name:

```
# send output to the default layout: layer 17, datatype 0
l.output(17, 0)

# send output to the default layout: layer named "OUT"
l.output("OUT")

# send output to the default layout: layer 17, datatype 0, named "OUT"
l.output(17, 0, "OUT")
```

Output can be sent to other layouts using the target function:

```
# send output to the second layout loaded:
target("@2")

# send output to "out.gds", cell "OUT_TOP"
target("out.gds", "OUT_TOP")
```

Output can also be sent to a report database using the report function:

```
# send output to a report database with description "Output database"
# - after the runset has finished this database will be shown
report("Output database")

# send output to a report database saved to "drc.lyrdb"
```

```
report("Output database", "drc.lyrdb")
```

When output is sent to a report database, the specification must include a formal name and optionally a description. The output method will create a new category inside the report database and use the name and description for that:

```
# specify report database for output
report("The output database")
...
# Send data from layer l to new category "check1"
l.output("check1", "The first check")
```

The report and target specification must appear before the actual output statements. Multiple report and target specifications can be present sending output to various layouts or report databases. Note that each report or target specification will close the previous one. Using the same file name for subsequent reports will not append data to the file but rather overwrite the previous file.

Layers that have been created using "output" can be used for input again, but care should be taken to place the input statement after the output statement. Otherwise the results will be unpredictable.

It is possible to open "side" reports and targets and send layers to these outputs without closing the default output.

To open a "side report", use new_report in the same way you use "report". Instead of switching the output, this function will return a new report object that can be included in the argument list of "output" for the layer that is to be sent to that side report:

```
# opens a new side report
side_report = new_report("Another report")
...
# Send data from layer l to new category "check1" to the side report
l.output(side_report, "check1", "The first check")
```

In the same way, "side targets" can be opened using new_target. Such side targets open a way to write certain layers to other layout files. This is very handy for debugging:

```
# opens a new side target for debugging
debug_out = new_target("debug.gds")
...
# Send data from layer l to the debug output, layer 100/0
l.output(debug_out, 100, 0)
```

Dimension specifications

Dimension specifications are used in many places: for coordinates, for spacing and width values and as length values. In all places, the following rules apply:

- Floating-point numbers are interpreted as micron values by default.
- Integer number are interpreted as database units by default (**Not** integer micron values!).
- To make explicitly clear what dimensions to use, you can add a unit.

Units are added using the unit methods:

- `0.1`: 0.1 micrometer
- `200`: 200 **database units**
- `200.dbu`: 200 database units
- `200.nm`: 200 nm

- `2.um` or `2.micron`: 2 micrometer
- `0.2.mm`: 0.2 millimeter
- `1e-5.m`: 1e-5 meter (10 micrometer)

Area units are usually square micrometers. You can use units as well to indicate an area value in some specific measurement units:

- `0.1.um2` or `0.1.micron2`: 0.1 square micron
- `0.1.mm2`: 0.1 square millimeter

Angles are always given in degree units. You can make clear that you want to use degree by adding the degree unit method:

- `45.degree`: 45 degree

Objects and methods

Runsets are basically scripts written in an object-oriented language. It is possible to write runsets that don't make much use of that fact, but having a notion of the underlying concepts will result in better understanding of the features and how to make full use of the capabilities.

In KLayout's DRC language, a layer is an object that provides a couple of methods. The boolean operations are methods, the DRC functions are methods and so on. Method are called "on" an object using the notation "object.method(arguments)". Many methods produce new layer objects and other methods can be called on those. The following code creates a sized version of the input layer and outputs it. Two method calls are involved: one sized call on the input layer returning a new layer object and one output call on that object.

```
input(1, 0).sized(0.1).output(100, 0)
```

The size method like other methods is available in two flavors: an in-place method and an out-of-place method. "sized" is out-of-place, meaning that the method will return a new object with the new content but not modify the object. The in-place version is "size" which modifies the object. Only the layer object is modified, not the original layer.

The following is the above code written with the in-place version:

```
layer = input(1, 0)
layer.size(0.1)
layout.output(100, 0)
```

Using the in-place versions is slightly more efficient in terms of memory since with the out-of-place version, KLayout will keep the unmodified copy as long as there is a chance it may be required. On the other hand the in-place version may cause strange side effects since because of the definition of the copy operation: a simple copy will just copy a reference to a layer object, not the object itself:

```
layer = input(1, 0)
layer2 = layer
layer.size(0.0)
layer.output(100, 0)
layer2.output(101, 0)
```

This code will produce the same sized output for layer 100 and 101, because the copy operation "layer2 = layer" will not copy the content but just a reference: after sizing "layer", "layer2" will also point to that sized layer.

That problem can be solved by either using the out-of-place version or by creating a deep copy with the "dup" function:

```
# out-of-place size:
layer = input(1, 0)
layer2 = layer
layer = layer.sized(0.0)
layer.output(100, 0)
```

```
layer2.output(101, 0)

# deep copy before size:
layer = input(1, 0)
layer2 = layer.dup
layer.size(0.0)
layer.output(100, 0)
layer2.output(101, 0)
```

Some methods are provided in different flavors including function-style calls. For example the width check can be written in two ways:

```
# method style:
layer.width(0.2).output("width violations")

# function style:
w = width(layer, 0.2)
output(w, "width violations")
```

The function style is intended for users not familiar with the object-oriented style who prefer a function notation.

Here is a brief overview over some of the methods available:

- Source, input and output:
  source, layout, cell, select, clip, input, labels, polygons, output, report, target
- DRC functions:
  width, space, separation (sep), notch, isolated (iso), enclosing (enc), enclosed, overlap
- Universal DRC (see below):
  drc
- Boolean operations:
  & (and), | (or), - (not), ^ (xor), + (join),
- Sizing:
  size, sized
- Merging:
  merge, merged
- Shape selections:
  covering, in, inside, interacting, outside, overlapping
  pull_inside, pull_interacting, pull_overlapping, These methods are available as in-place operations as well:
  select_covering, select_interacting, select_inside, select_outside, select_overlapping
- Filters:
  rectangles, rectilinear, with_area, with_bbox_height, with_bbox_width, with_bbox_max, with_bbox_min, with_perimeter
  with_angle, with_length
  These methods are available as version selecting the opposite:
  non_rectangles, non_rectilinear, without_area, without_bbox_height, without_bbox_width, without_bbox_max, without_bbox_min,
  without_perimeter without_angle, without_length
- Text filters:
  texts, texts_not,
- Transformations:
  moved, rotated, scaled, transformed
  These methods are available as in-place versions as well: move, rotate, scale, transform
- Polygon manipulations:
  extents, hulls, holes

- Edge manipulations:
centers, end_segments, start_segments, extended, extended_in, extended_out
- Information:
length, perimeter, area, polygons?, edges?, edge_pairs?, is_box?, is_clean?, is_empty?, is_merged?, is_raw?
- Layer mode:
raw, clean
- Layer type conversions:
edges, first_edges, second_edges, polygons

Polygon and edge layers

KLayout knows four layer types: polygon, edge, edge pair and text layers. Polygon and edge layers are the basic layer types for geometrical operations.

Polygon layers are created from original layers using input or polygons. "input" will also turn texts into small polygons with a size of 2x2 DBU while "polygons" will skip texts. For handling texts, the labels method is recommended which renders a true text layer. Text layers are described below.

Polygon layers describe objects having an area ("filled objects" in the drawing view). Such objects can be processed with boolean operations, sized, decomposed into holes and hull, filtered by area and perimeter and so on. DRC methods such as width and spacing checks can be applied to polygons in a different way than between different polygons (see space, separation and notch for example).

Polygons can be raw or merged. Merged polygons consist of a hull contour and zero to many hole contours inside the hull. Merging can be ensured by putting a layer into "clean" mode (see clean, clean mode is the default). Raw polygons usually don't have such a representation and consist of a single contour folding inside to form the holes. Raw polygons are formed in "raw" mode (see raw).

Egde layers can be derived from polygon layers and allow the description is individual edges ("sides") of a polygon. Edge layers offer DRC functions similar for polygons but in a slightly different fashion - edges are checked individually, non considering the polygons they belong to. Neither do other parts of the polygons shield interactions, hence the results may be different.

Edges can be filtered by length and angle. extended allows erecting polygons (typically rectangles) on the edges. Edge layers are useful to perform operations on specific parts of polygons, for example width or space checks confined to certain edge lengths.

Edges do not differentiate whether they originate from holes or hulls of the polygon. The direction of edges is always following a certain convention: when looking from the start to the end point of an edge, the "inside" of the polygons from which the edges were derived, is to the right. In other words: the edges run along the hull in clockwise direction and counterclockwise along the holes.

Merged edges are joined, i.e. collinear edges are merged into single edges and degenerate edges (single-point edges are removed). Merged edges are present in "clean" mode (see clean, clean mode is the default).

Polygons can be decomposed into edges with the edges method. Another way to generate edges is to take edges from edge pair objects which are generated by the DRC check functions.

Text collections

Starting with version 0.27, KLayout offers support for text layers. "Texts" are basically locations with a label, i.e. a dot with an arbitrary string attached. "Text collections" are collections of such objects.

Texts can be used to select polygons or as net names in net extractions.

Text collections are kept in "text layers". These are created using the labels methods instead of "input".

These operations are supported for text layers:

- Boolean AND with a polygon layer: will select those texts which are inside or at the border of a polygon. interact is a synonym for this operation.
- Boolean NOT with a polygon layer: will select those texts which are outside of any polygon. not_interact is a synonym for this operation.
- As second layer for region interact: this way, polygons can be selected which are tagged with certain texts.
- Text filtering by string: texts can be filtered either by matching against a fixed text or a glob pattern. The methods provided for this purpose are: texts and texts_not

- flatten will flatten the hierarchy of a text layer.
- Polygon or edge generation around the text's location: polygons and edges

Edge pairs and edge pair collections

Edge pairs are objects consisting of two edges. Edge pairs are handy when discribing a DRC check violation, because a violation occurs between two edges. The edge pair generated for such a violation consists of the parts of both edges violation the condition. For two-layer checks, the edges originate from the original layers - edge 1 is related to input 1 and edge 2 is related to input 2.

Edge pair collections act like normal layers, but very few methods are defined for those. Edge pairs can be decomposed into single edges (see edges, first_edges and second_edges).

Edge pairs can be converted to polygons using polygons. Edge pairs can have a vanishing area, for example if both edges are coincident. In order to handle such edge pairs properly, an enlargement can be applied optionally. With such an enlargement, the polygon will cover a region bigger than the original edge pair by the given enlargement.

Raw and clean layer mode

KLayout's DRC engine supports two basic ways to interpret geometrical information on a layer: in clean mode, polygons or edges are joined if they touch. If regions are drawn in separate pieces they are effectively joined before they are used. In raw mode, every polygon or shape on the input layer is considered a separate part. There are applications for both ways of looking at a set of input shapes, and KLayout supports both ways.

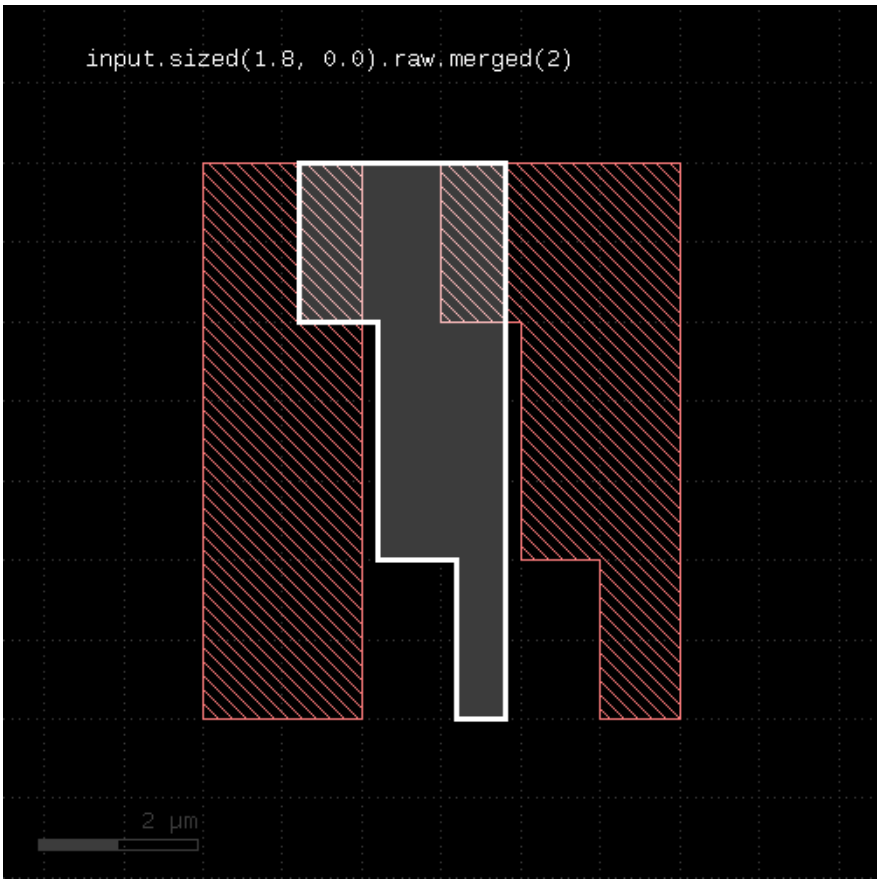
Clean mode is the default - every layer generated or taken from input will be used in clean mode. To switch to raw mode, use the "raw" method. "raw mode" is basically a flag set on the layer object which instructs the engine not the merge polygons prior to use. The raw mode flag can be reset with the "clean" method.

Most functions implicitly merge polygons and edges in clean mode. In the documentation this fact is referred to as "merged semantics": if merged semantics applies for the function, coherent polygons or edges are considered one object in clean mode. In raw mode, every polygon or edge is treated as an individual object.

One application is the detection of overlapping areas after a size step:

```
overlaps = layer.size(0.2).raw.merged(2)
```

That statement has the following effect:

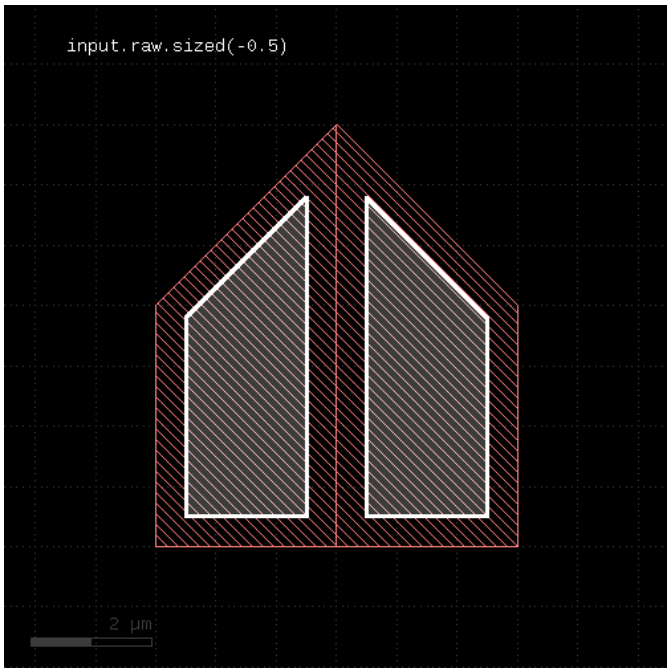


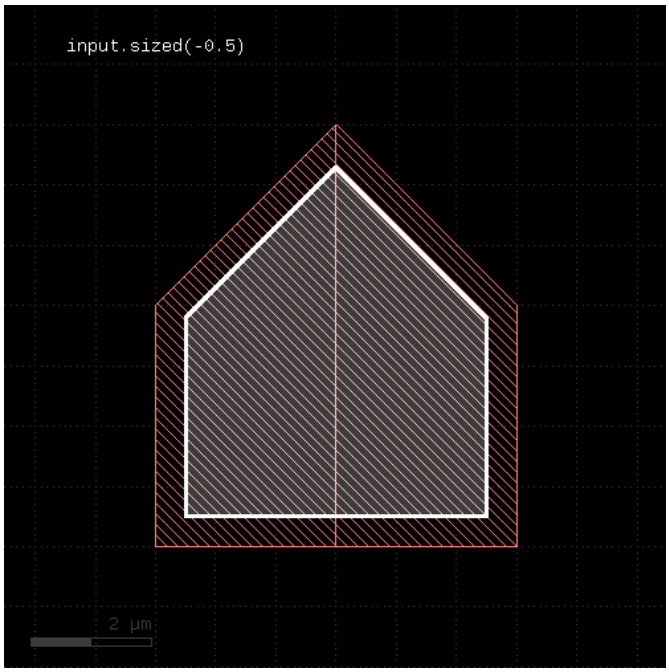
The "merged" method with an argument of 2 will produce output where more than two polygons overlap. The size function by default creates a clean layer, but separate polygons for each input polygon, so by using "raw", the layer is switched into raw mode that makes the individual polygons accessible without merging them into one bigger polygon.

Please note that the raw or clean methods modify the state of a layer so beware of the following pitfall:

```
layer = input(1, 0)
layer.raw.sized(0.1).output(100, 0)

# this check will now be done on a raw layer, since the
# previous raw call was putting the layer into raw mode
layer.width(0.2).ouput(101, 0)
```

The following two images show the effect of raw and clean mode:

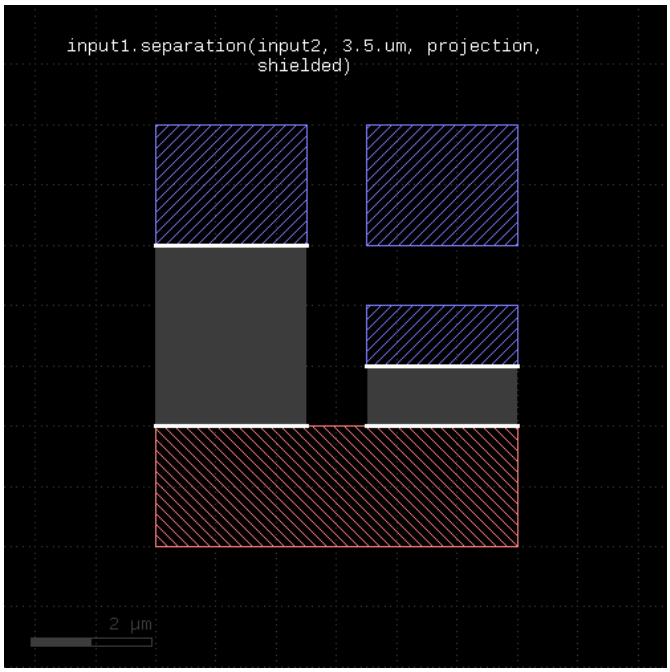


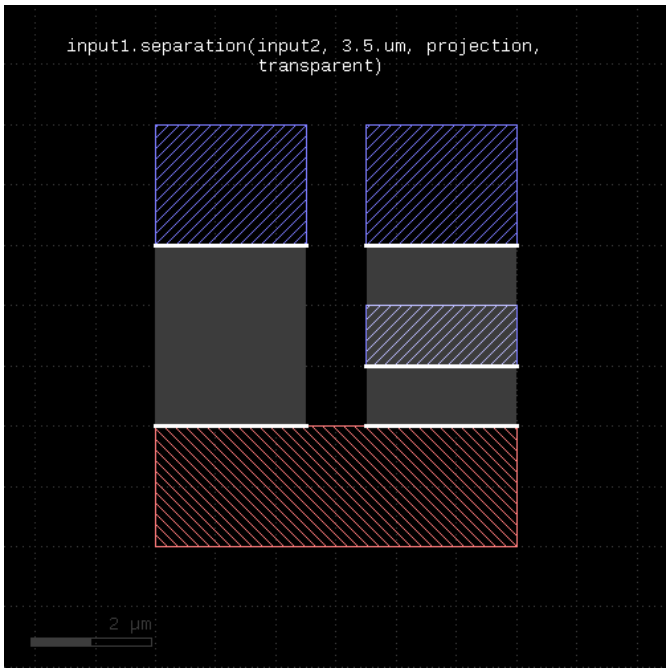


Shielding

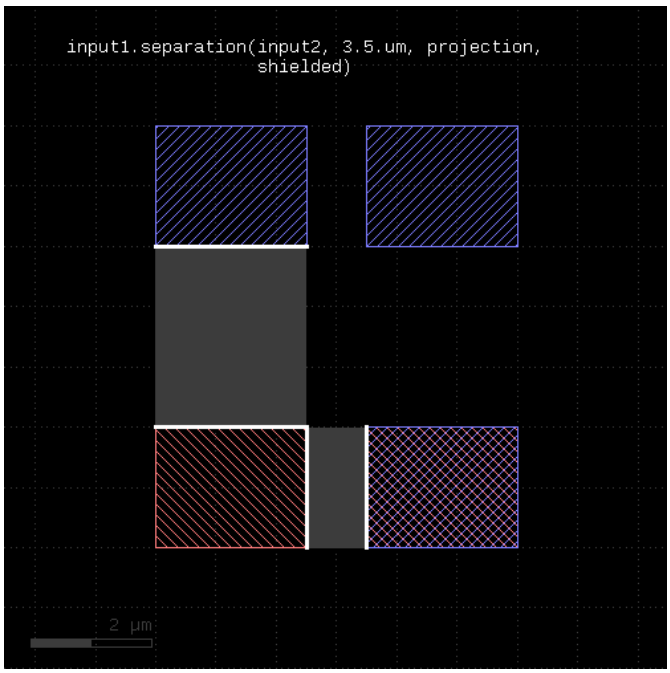
"Shielding" is a concept where DRC measurements do not "look through" layout features. With shielding, a DRC violation is skipped when another feature would (fully) block the violation marker's path. Shielding is available and enabled by default for the (internal or external) distance-based DRC functions: width, space, separation (sep), notch, isolated (iso), enclosing (enc) or enclosed or overlap. Shielding is turned off using the "transparent" option or turned on using "shielded". The latter is only for clarity, but is not required as shielding is enabled by default.

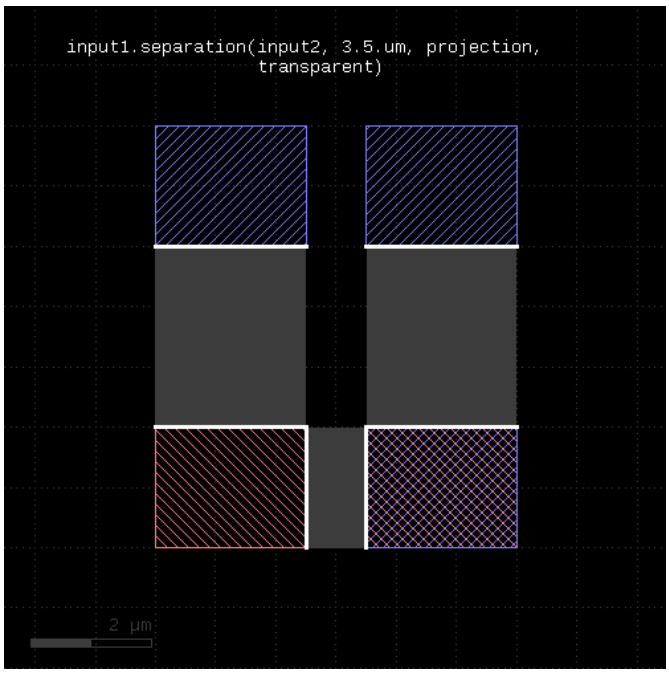
The following examples demonstrate the effect of shielding: in the right example, shielding is turned off. Hence, the violation between the upper box on the right and the lower bar is no longer shielded by the small bar between them and this additional violation is reported too.





Although shielding feels more natural, it can have an adverse effect as it is effective at zero distance already. In the following example, the second layer is a subset of the first. When testing the distance between second and first, the overlapping first layer shapes will block the separation measurement in shielded mode. Hence, only transparent mode will render the actual distance violation. The bottom right blue box is not shielded by the overlaying red box:





Universal DRC

Starting with version 0.27, the DRC language got a new feature which is "universal DRC".

On one hand, this is a more convenient way to write DRC checks because it allows specifications using natural compare operators. For example, the following plain width check

```
...
drc_w = input(1, 0).width(0.2)
...
```

can be written as:

```
...
drc_w = input(1, 0).drc(width < 0.2)
...
```

The drc method is the "universal DRC" method. It takes an operator. In the simple case, this operator is a simple constraint of the form "measurement < value", but it supports a number of different variants:

- `drc( measurement != value )`: renders markers where the dimension is not matching the value
- `drc( measurement == value )`: renders markers where the dimension is matching the given value
- `drc( measurement less_op value )`: ("less_op" is < or <=) renders markers where the dimension is less than or less or equal to the given value
- `drc( measurement greater_op value )`: ("greater_op" is > or >=) renders markers where the dimension is greater than or greater or equal to the given value
- `drc( lower_value less_op measurement greater_op upper_value )`: renders markers where the dimension is between the lower and upper value

"measurement" is "width", "notch", "isolated" ("iso"), "separation" ("sep"), "overlap", "enclosed" or "enclosuring" ("enc"). The last three checks are two-layer checks which require a second layer. The second layer is specified together with the measurement like this:

```
...
l1 = input(1, 0)
l2 = input(2, 0)
drc_sep = l1.drc(separation(l2) <= 0.5)
...
```

Options are also specified together with the measurement and follow the same notation as the plain methods. For example to specify "projection" metrics, use:

```
...
drc_w = input(1, 0).drc(width(projection) < 0.2)
...
```

However, the universal DRC is much more than a convenient way to write checks: it offers a number of options to further process the results. The functionality behind the universal DRC function is basically a kind of loop over all primary shapes (the ones from the layer the "drc" function is called on). The operations in the drc function's brackets is executed on each of the primary shapes where the neighborhood of that single shape is considered. This scheme is more efficient and enables applications beyond the capabilities of the plain layer methods.

For example, the boolean "&" operator implements a "local" boolean AND inside this loop. This allows to efficiently check for both space and width violations:

```
...
drc_ws = input(1, 0).drc((width < 0.2) & (space < 0.3))
```

```
...
```

The boolean AND is computed between the edges on the primary shape and returns the parts where both space and width violations are flagged. The boolean operation is more efficient than the plain alternative:

```
...
drc_ws1 = input(1, 0).width(0.2).edges
drc_ws2 = input(1, 0).space(0.3).edges
drc_ws = drc_ws1 & drc_ws2
...
```

The reason is that performing the boolean computation in the local loop can be shortcut if one inputs is empty. It does not need to store a (potentially big) edge set with edges as produced by the plain-method implementation. Instead it will work with a temporary and local edge set only and free the memory space as soon as it moves on to the next primary shape.

Overall, the universal DRC function is a rich feature and offers filters based on polygons or edge properties, polygon or edge manipulation operators, conditionals and a lot more. For more details see the drc function documentation.

Logging and verbosity

While the runset is executed, a log is written that lists the methods and their execution times. The log is enabled using the verbose function. The log and info functions allows entering additional information into the log. "info" will enter the message if verbose mode is enabled. "log" will enter the message always. silent is equivalent to "verbose(false)".

The log is shown in the log window or - if the log window is not open - on the terminal on Linux-like systems.

The log function is useful to print result counts during processing of the runset:

```
...
drc_w = input(1, 0).width(0.2)
log("Number of width violations: #{drc_w.data.size}")
...
```

The error function can be used to output error messages unconditionally, formatted as an error. The log can be sent to a file instead of the log window or terminal output, using the log_file function:

```
log_file("drc_log.txt")
verbose(true)
info("This message will be sent to the log file")
...
```

The profile function will collect profiling information during the DRC run. At the end of the script, the operations are printed to the log output, sorted by their CPU time and approximate memory footprint. "profile" can be given a numerical argument indicating the number of operations to print. Lower-ranking operations are skipped in that case. By default, all operations are printed.

```
# enables profiling
profile
...
```

The tiling option

Tiling is a method to reduce the memory requirements for an operation. For big layouts, pulling a whole layer into the engine is not a good idea - huge layouts will require a lot of memory. The tiling method cuts the layout into tiles with a given width and height and processes them individually. The tiling implementation of KLayout can make use of multiple CPU cores by distributing the jobs on different cores.

Tiling does not come for free: some operations have a potentially infinite range. For example, selecting edges by their length in clean mode basically requires to collect all pieces of the edge before the full length can be computed. An edge running over a long length however may cross multiple tiles, so that the pieces within one tile don't sum up to the correct length.

Fortunately, many operations don't have an infinite range, so that tiling can be applied successfully. These are the boolean operations, sizing and DRC functions. For those operations, a border is added to the tile which extends the region inside which the shapes are collected. That way, all shapes potentially participating in an operation are collected. After performing the operation, polygons and edges extending beyond the tile's original boundary are clipped. Edge pairs are retained if they touch or overlap the original tile's border. That preserves the outline of the edge pairs, but may render redundant markers in the tile's border region.

For non-local operations such as the edge length example, a finite range can be deduced in some cases. For example, if small edges are supposed to be selected, the range of the operation is limited: longer edges don't contribute to the output, so it does not matter whether to take into account potential extensions of the edge in neighboring tiles. Hence, the range is limited and a tile border can be given.

To enable tiling use the tiles function. The threads function specifies the number of CPU cores to use in tiling mode. flat will disable tiling mode:

```
# Use a tile size of 1mm
tiles(1.mm)
# Use 4 CPU cores
threads(4)

... tiled operations ...

# Disable tiling
flat

... non-tiled operations ...
```

Some operations implicitly specify a tile border. If the tile border is known (see length example above), explicit borders can be set with the tile_borders function. no_borders will reset the borders (the implicit borders will still be in place):

```
# Use a tile border of 10 micron:
tile_borders(10.um)

... tile operations with a 10 micron border ...

# Disable the border
no_borders
```

A word about the tile size: typically tile dimensions in the order of millimeters is sufficient. Leading-edge technologies may require smaller tiles. The tile border should not be bigger than a few percent of the tile's dimension to reduce the redundant tile overlap region. In general using tiles is a compromise between safe function and performance. Very small tiles imply some performance overhead do to shape collection and potentially clipping. In addition, the clipping at the tile's borders may introduce artificial polygon nodes and related snapping to the database unit grid. That may not be desired in some applications requiring a high structure fidelity. Hence, small tiles should be avoided in that sense too.

Hierarchical mode

Alternatively to the tiling option, hierarchical mode is available. In hierarchical mode, the DRC functions operate on subcells if their configuration allows this. The algorithm will automatically detect whether an operation can be performed in a subcell. For example, a sizing operation can be done inside a subcell, if the cell's content is not connected to anything outside the cell.

To enable hierarchical operations, use the "deep" statement:

```
report("deep 2")

# enable deep (hierarchical) operations
deep

poly = input(3)
```

```
spc = poly.space(0.5)
spc.output("poly space >0.5")
```

"deep" is not compatible with tiling. "tiles" will disable "deep" and vice versa. To disable deep mode, use "flat".

Deep processing is a layer property. After "deep" has been specified, layers derived with "input" are declared to be deep - i.e. hierarchical operations are enabled on them. Operations on deep layers will usually render other deep layers. This is also true for edge and edge pair layers. For example, the "space" operation above will render a hierarchical edge pair layer.

In binary operations such as boolean operations, the operation is performed hierarchically, if both involved layers are deep. A layer can be explicitly converted to a flat layer using "flatten".

To check whether a layer is deep, use "is_deep?".

```
report("deep 2")

# enable deep (hierarchical) operations
deep

poly = input(3)
puts poly.is_deep?   # -> true
poly.flatten
puts poly.is_deep?   # -> false
```

Most operations are hierarchy enabled, with a few exceptions. Some operations - specifically the transformation operations such as "move", "rotate" and the anisotropic sizing or the grid snap operations will generate cell variants. Such variants reflect different configurations of cells with respect to the requested operation. For example, with anisotropy (x != y), rotated cells need to be treated differently from non-rotated ones. In the "snap" feature, cell variants are created if the cell's instances are not all on-grid. Most functions need to create variants only when the same cell is instantiated with different magnification factors.

When writing back a layout with cell variants, new versions of cells will appear.

When sending the output of hierarchical operations to a report database, the markers will be listed under the cell name they appear. A sample cell instance is provided within the marker database to allow visualizing the marker in at least one context.

Limitations

Functions which require merged polygons utilize the net clustering algorithm to form the merged polygons. All connected shapes are collected and merged into a bigger polygon. This happens in the lowest possible level on the hierarchy where the shape clusters are complete. In some cases - when the shapes come from big coherent regions - this may happen on the top cell level and the resulting polygon will be very big. This will lead to poor performance.

The DRC's hierarchical mode will - except for cell variants in the cases mentioned - not modify the cell hierarchy. This hierarchy-preserving nature is good for certain applications, but leads to a compromise in terms of resolving hierarchically different configurations. As the algorithm is not allowed to create variants in most cases, the only remaining option is to propagate results from such cases into the parent cells. In the worst case this will lead to flattening of the layout and loss of hierarchy.

DRC and user properties

The DRC feature has some support for user properties. User properties are sets of key/value pairs attached to shapes. This is a standard feature of KLayout and GDS/OASIS. The GDS format supports numerical (positive integer) keys and string values while OASIS supports more types of keys and values - specifically string keys.

For DRC, the property set attached to a shape is regarded as a whole. The DRC can act on these properties in specific ways:

- **Ignore properties**: this is the default mode
- **Use properties**: configure operations such as checks and some boolean operations to only consider shapes with the same or different properties
- **Map/filter**: change property keys/sets or remove them
- **Transfer properties**: through operations, e.g. "size" or "with_..." selectors

Specifically, DRC functions can also **generate** properties. Currently there is only the "nets" method which attaches net identity information to shapes involved in a "connect" statement. This feature opens a path to implementing "connected" or "unconnected" mode boolean operations and DRC checks.

As of this writing, user property support is somewhat experimental. User properties support has a huge potential, so there is more to come.

Currently, the following operations can be conditioned to act on shapes with same or different properties:

- Polygon boolean operations: "and", "not" and "andnot"
- Polygon-mode DRC checks such as "separation", "space", "isolated" etc.

A variety of operations can transfer properties, i.e. edge-pair-to-polygon, edge-pair-to-edges, polygon-to-edges, edge-to-polygon, some filters, the "size" function. It is planned to enable most features to transfer properties where applicable.

Property generation is supported currently by:

- Reading properties from input layouts
- Using the "nets" feature to generate net identity properties.

Property manipulation is supported in a very basic way: properties can be removed entirely from a layer or certain property keys can be selected and optionally mapped to a different key. Property values cannot be manipulated currently.

In general, once a layer has properties, shapes with different properties are regarded as non-interacting. When shapes are merged, only groups of shapes with the same properties are merged into bigger chunks. This applies to polygons and edges. This can have the strange consequence that after merge, still polygons may overlap. Note that this only applies to the case with properties. The normal behavior is not changed.

Reading user properties

By default, user properties are not read into the shape containers. You need to enable them explicitly:

```
l1 = input(1, 0, enable_props)
```

At this point you can select certain keys from the set of properties. For example to select only values with key 17 and 18 (numerical), use:

```
l1 = input(1, 0, select_props(17, 18))
```

You can also select and map keys to other keys, like this:

```
l1 = input(1, 0, map_props({ 17 => 1, 18 => 18 }))
```

This will map values with key 17 to 1 and read those from 18 while maintaining the key. Values with other keys are ignored. See input function documentation for more details.

Manipulating properties

Once you have a layer with properties, you can remove them:

```
layer_without_properties = layer.remove_props
```

You can also apply select_props or map_props to filter values with certain keys or map keys:

```
reduced_layer = layer.select_props(17, 18)
reduced_layer = layer.map_props({ 17 => 1, 18 => 18 })
```

Generating properties as net identities

The most important application is to use the nets method to generate net identity properties:

```
connect(metal1, via1)
connect(via1, metal2)

metal1_nets = metal1.nets
```

By default, a unique net identifier (a tuple of circuit and net name) is generated on property key 0. You can specify the key as well:

```
metal1_nets = metal1.nets(prop(17))
```

The "nets" function has a number of options, specifically you can filter certain nets (by name or circuit + name). This makes the "nets" function useful for other purposes too. If you do not need properties then, specify "nil" as the property key:

```
metal1_vdd_net = metal1.nets(prop(nil), "VDD")
```

Using properties in checks and boolean operations

The main purpose of properties is to use them in operations. To confine a boolean operation to shapes with different properties, use the props_ne keyword. To confine a boolean operation to shapes with the same properties, use props_eq:

```
connect(metal1, via1)
connect(via1, metal2)

metal1_nets = metal1.nets
metal2_nets = metal2.nets

metal1_over_metal2_connected = metal1_nets.and(metal2_nets, props_eq)
metal1_over_metal2_unconnected = metal1_nets.and(metal2_nets, props_ne)
```

You can also instruct this operation to emit the original properties on the output with props_copy:

```
result_with_props = metal1_nets.and(metal2_nets, props_eq + props_copy)
```

Similarly, properties can participate in checks:

```
connect(metal1, via1)
connect(via1, metal2)

metal1_nets = metal1.nets
metal2_nets = metal2.nets

metal1_space_connected = metal1_nets.space(0.4.um, props_eq)
metal1_space_unconnected = metal1_nets.space(1.um, props_ne)
```

"props_eq", "props_ne" and "props_copy" are also available on the generic DRC function (drc), which opens new options, e.g. detecting potential short locations ("critical area") between unconnected nets:

```
connect(metal1, via1)
connect(via1, metal2)

metal1_nets = metal1.nets
metal2_nets = metal2.nets

critical_area = l1_nets.drc(primary.sized(0.2.um) & foreign.sized(0.2.um), props_ne)
```

1.5. Layout vs. Schematic (LVS)

LVS is a verification step which checks whether a layout matches the circuit from the schematic. The LVS feature is described in the following topic chapters:

- Layout vs. Schematic (LVS) Overview
- LVS Introduction
- LVS Devices
- LVS Device Classes
- LVS Device Extractors
- LVS Input/Output
- LVS Connectivity
- LVS Compare
- LVS Netlist Tweaks

A reference for the functions and objects available for LVS scripts can be found here: LVS Reference.

1.5.1. Layout vs. Schematic (LVS) Overview

Basic usage of LVS scripts

Starting with version 0.26, KLayout supports LVS as a built-in feature. LVS is an important step in the verification of a layout: it ensures the drawn circuit matches the desired schematic.

The basic functionality is simply to analyze the input layout and derive a netlist from this. Then compare this netlist against a reference netlist (schematic). If both netlist are equivalent, the circuit is likely to work in the intended fashion.

Beside the layout, a LVS script will also need a schematic netlist. Currently, KLayout can read SPICE-format netlists. The reader can be configured to some extent, so the hope is that a useful range of SPICE netlists can be digested.

While the basic idea is simple, the details become pretty complex. This documentation tries to cover the solutions KLayout offers to implement LVS as well as the constraints imposed by this process.

KLayout's LVS is integrated into the Macro Development IDE the same way as DRC scripts. In fact, LVS is an add-on to DRC scripts. All DRC functions are available within LVS scripts. Netlist extraction is performed in the DRC framework which was given the ability to recognize devices and connections and turn them into a netlist. Although DRC does not really benefit from these extensions, they are still useful for implementing Antenna checks for example. As it happens, the majority of features required for LVS is documented in the DRC Reference, while the few add-ons required specifically for LVS are documented in LVS Reference.

LVS scripts are created, edited and debugged in the Macro Editor IDE. They are managed in the "LVS" tab. For more details about the IDE, see About Macro Development. For an introduction about how to work with DRC scripts see Design Rule Checks (DRC) Basics.

LVS scripts carry the ".lylvs" extension for the XML form (in analogy to ".lydrc" for DRC) and ".lvs" for the plain text form (same as ".drc"). Like DRC scripts, LVS scripts can be executed standalone in batch mode like DRC scripts. See "Using KLayout as a standalone DRC engine" in Design Rule Checks (DRC) Basics.

KLayout's LVS implementation

The LVS implementation inside KLayout is designed to be highly flexible in terms of connectivity, device recognition and input/output channels. Here are some highlights:

- **Agnostic approach:** KLayout tries to make as few assumptions as possible. It does not require labels (although they are helpful), a specific hierarchy, specific cell names or specific geometries. Netlist extraction is done purely from the polygons of the layout. Labels and the cell hierarchy add merely useful hints which simplify debugging and pin assignment, but no strict requirement.

- **Hierarchical analysis:** KLayout got a hierarchical layout processing engine to support hierarchical LVS. Hierarchical processing means that boolean operations happen inside the local cell environment as far as possible. As a consequence, devices are recognized inside their layout cell and layout cells are turned into respective subcircuits in the netlist. The netlist compare will benefit as it is able follow the circuit hierarchy. This is more efficient and gives better debugging information in case of mismatches. As a positive side effect of hierarchical layout processing the runtimes for some boolean and other operations is significantly reduced in most cases.

- **Hierarchically stable:** KLayout won't modify the layout's hierarchy nor will it introduce variants - at least for boolean and some other operations. This way, matching between layout and schematic hierarchy is maintained even after hierarchical DRC operations. Variants are introduced only for some anisotropic operations, the grid snap method and some other features which require differentiation of cells in terms of location and orientation.

- **Flexible engine:** The netlist formation engine is highly flexible with respect to device recognition and connectivity extraction. First, almost all DRC features can be used to derive intermediate layers for device formation and connectivity extraction. Second, the device recognition can be scripted to implement custom device extractors. Five built-in device extractors are available for MOS and bipolar transistors, resistors, capacitors and diodes.

- **Flexible I/O:** Netlists are KLayout object trees and their components (nets, devices, circuits, subcircuits ...) are fully mapped to script objects (for the main class see Netlist in the API documentation). Netlists can therefore be analyzed and manipulated within LVS scripts or in other contexts. It should be possible to fully script readers and writers for custom formats. Netlists plus the corresponding layout elements (sometimes called "annotated layout") can be persisted in a KLayout-specific, yet open format. SPICE format is available to read and write pure netlist information. The SPICE reader and writer is customizable through delegate classes which allow tailoring of the way devices are read and written.

- **User interface integration:** KLayout offers a browser for the netlist extraction results and LVS reports (cross-reference, errors).

Terminology

KLayout employs a specific terminology which is explained here:

- **Circuit:** A graph of connected elements as there are: devices, pins and subcircuits. The nodes of the graph are the nets connecting at least two elements. If derived from a layout, a circuit corresponds to a specific layout cell.
- **Abstract circuits:** Abstract circuits are circuits which are cleared from their inner structure. Such circuits don't have nets and define pins only. Abstract circuits are basically "black boxes" and LVS is required to consider their inner structure as "don't care". Abstract circuits are useful to reduce the netlist complexity by taking out big IP blocks verified separately (e.g. RAM blocks).
- **Pin:** A point at which a circuit makes a connection to the outside. Circuits can embed other circuits as "subcircuits". Nets connecting to the pins of these subcircuits will propagate into the subcircuit and connect further elements there. Pins are usually attached to one net - in some cases, pins can be unattached (circuits abstracts). Pins can be named. Upon extraction, the pin name is derived from the name of the net attached to the pin.
- **Subcircuit:** A circuit embedded into another circuit. One circuit can be used multiple times, hence many subcircuits can reference the same circuit. If derived from a layout, a subcircuit corresponds to a specific cell instance.
- **Device:** A device is a n-terminal entity describing an atomic functional unit. Devices are passive devices (resistors, capacitors) or active devices such as transistors.
- **Device class:** A device class is a type of device. Device classes are of a certain kind and there can be multiple classes per type. For example for MOS transistors, the kind is "MOS4" (a four-terminal MOS transistor) and there is usually "NMOS" and "PMOS" classes at least in a CMOS process. A device class typically corresponds to a model in SPICE.
- **Device extraction:** Device extraction is the process of detecting devices and forming links between conductive areas and the device bodies. These links will eventually form the device terminals.
- **Device combination:** Device combination is the process of forming single devices from combinations of multiple devices of the same class. For example, serial resistors can be combined into one. More importantly, parallel MOS transistors ("fingered" transistors) are combined into a single device. Device combination is a step explicitly requested in the LVS script.
- **Terminal:** A "terminal" is a pin of a device. Terminals are typically named after their function (e.g. "G" for the gate of a MOS transistor).
- **Connectivity:** The connectivity is a description of conductive regions in the technology stack. A layer has intra-layer and inter-layer connectivity: "Intra-layer connectivity" means that polygons on the same layer touching other polygons form a connected - i.e. conductive - region. "Inter-layer connectivity" means that two layers form a connection where their polygons overlap. The sum of these rules forms the "connectivity graph".
- **Netlist:** A hierarchical structure of circuits and subcircuits. A netlist typically has a top circuit from which other circuits are called through subcircuits.
- **Extracted netlist:** The extracted netlist is the netlist derived from the layout. Sometimes, "extracted netlist" describes the netlist enriched with parasitic elements such as resistors and capacitors derived from the wire geometries. In the context of KLayout's LVS, "extracted netlist" is the pure connectivity without parasitic elements.
- **Schematic:** The "schematic" is a netlist taken as reference for LVS. The "schematic" is thought of the "drawn" netlist that is turned into a layout by the physical implementation process. In LVS, the layout is turned back into the "extracted netlist" which is compared to the schematic.
- **Annotated layout, Net geometry:** The collection of polygons belonging to the individual nets. Each net inside a circuit is represented by a bunch of polygons representing the original wire geometry and the device terminals. As nets can propagate to subcircuits through pins, nets and therefore annotated layout carries a per-net hierarchy. The per-net hierarchy consists of the subcircuits attached to one net and the nets within these subcircuits that connect to the outer net. Subcircuits can instantiate other subcircuits, so the hierarchy may extend over many levels.

- **Layout to netlist database (L2N DB):** This is a data structure combining the information from the extracted netlist and the annotated layout into a single entity. The L2N database can be used to visualize nets, probe nets from known locations and perform other analysis and manipulation steps. An API for handling L2N databases is available.
- **Cross reference:** The cross reference is a list of matching objects from the two netlists involved in a LVS netlist compare ("pairing"). The cross-reference also lists non-matching items and inexact pairs. "Inexact pairs" are pairs of objects which do not match precisely, but still are likely to be paired. The cross reference also keeps track of the compare status - i.e. whether the netlists match and if not, where a mismatch originates from.
- **LVS database:** The "LVS database" is the combination of L2N database, the schematic netlist and the cross-reference. It's a complete image of the LVS results. An API is available to access the elements of the LVS database.
- **Labels:** "Labels" are text objects drawn in a layout to mark certain locations on certain layers with a text. Typically, labels are used to assign net names - if included in the connectivity, nets formed from such labels get a name according to the text string of the label.

1.5.2. LVS Introduction

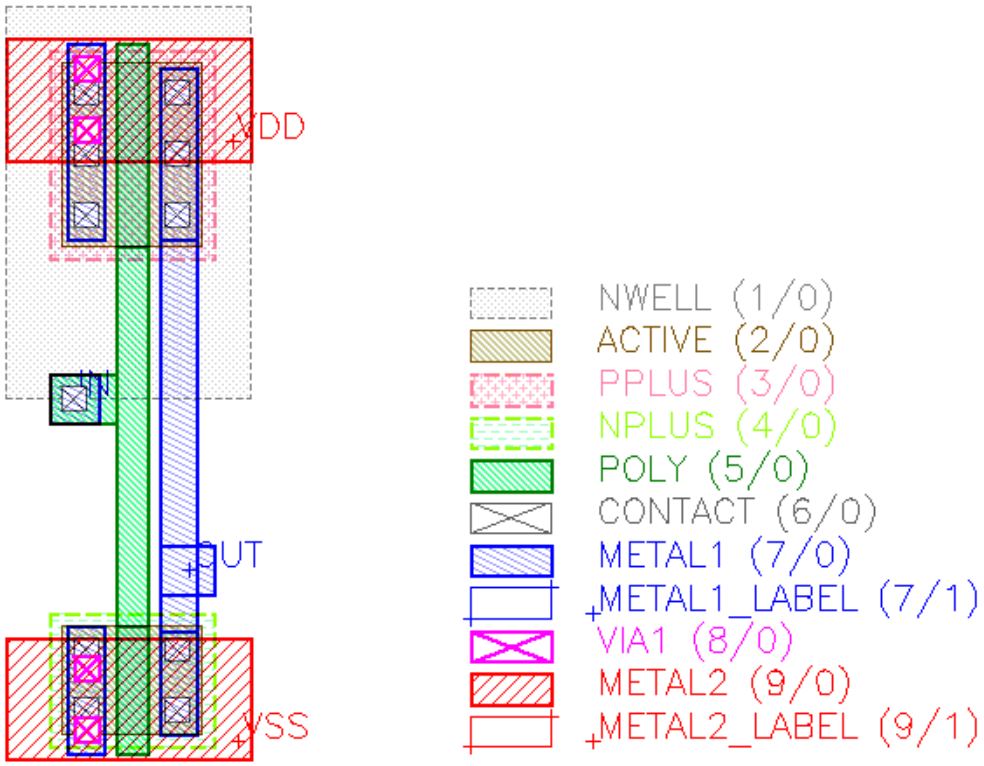
LVS introduction

For introducing the LVS feature we consider the most simple CMOS structure there is: the two-transistor inverter.

Layout

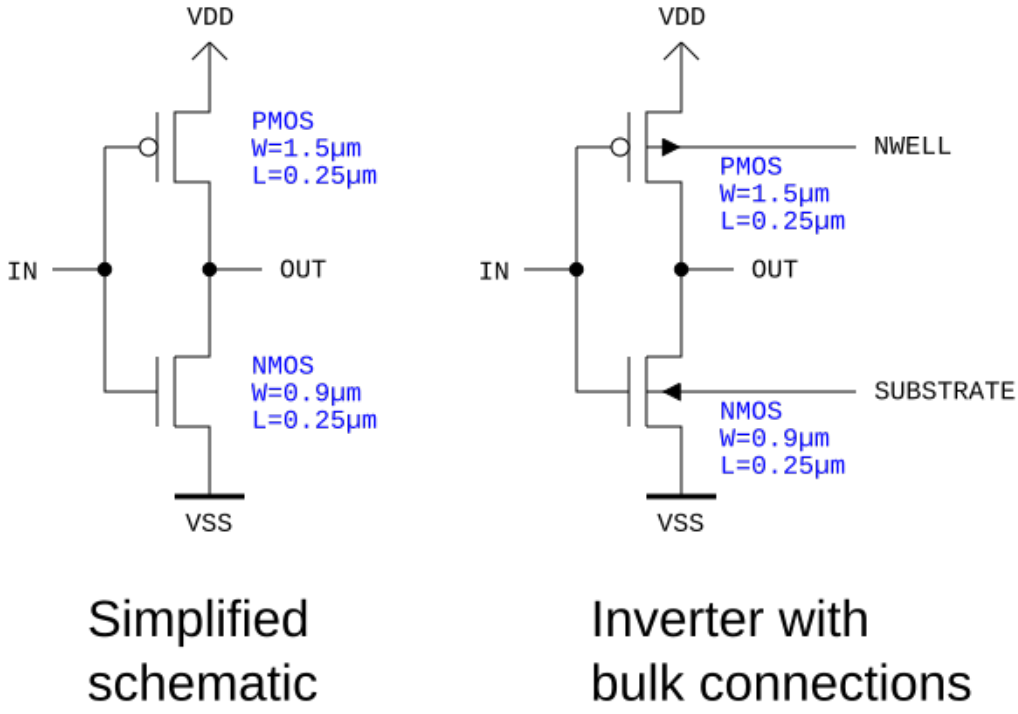
The inverter consists of two MOS transistors. A single transistor is made from an active region (a rectangle on the ACTIVE layer) and a gate (POLY layer) crossing the active region. The gate forms the channel from source to drain regions (left and right of gate). Contacts (CONTACT) provide connections from the first metal layer (METAL1) to the gate polysilicon (POLY) and to source/drain regions (where over ACTIVE). Via holes (VIA1) provide connections from the first (METAL1) to the second metal (METAL2). Finally, specific devices are formed by the source/drain implants which is n+ (NPLUS marker) for NMOS and p+ (PPLUS marker) for PMOS devices. PMOS devices sit in a n implant region (n-well) which forms the p-channel region. NMOS devices are built over substrate which is p doped to supply the n-channel region.

The actual layout is made as a standard cell. Multiple standard cells can be arrayed horizontally in a row. The power rails are formed in the second metal for VDD at the top and VSS at the bottom. The n-well extends over the top of the cell and is supposed to connect to neighbor well regions:



Schematic

For the inverter we can draw a schematic in a simplified form (left) and in a more realistic form (right) which also includes the bulk potentials of the transistors. It is important to keep the bulk of of the transistors at a defined potential to avoid latch-up. Hence we need pins for these terminals too. This makes a total of six pins: for input (IN) and output (OUT), for the power (VDD, VSS) and the two bulk potentials (NWELL, SUBSTRATE):



For LVS we first need a reference schematic. This is the SPICE netlist corresponding to the schematic with the bulk connections:

```
* Simple CMOS inverer circuit (inv.cir)
.SUBCKT INVERTER VSS IN OUT NWELL SUBSTRATE VDD
Mp VDD IN OUT NWELL     PMOS W=1.5U L=0.25U
Mn OUT IN VSS SUBSTRATE NMOS W=0.9U L=0.25U
.ENDS
```

The circuit we are going to analyze is a cell which is embedded in bigger circuits. Hence it makes sense to describe the inverter as a subcircuit. If the netlist consists of a subcircuit only, KLayout will consider this circuit. Otherwise it will consider the global definitions as the main circuit. In the latter case, pins cannot be defined while with subcircuits pins can be listed as given names too.

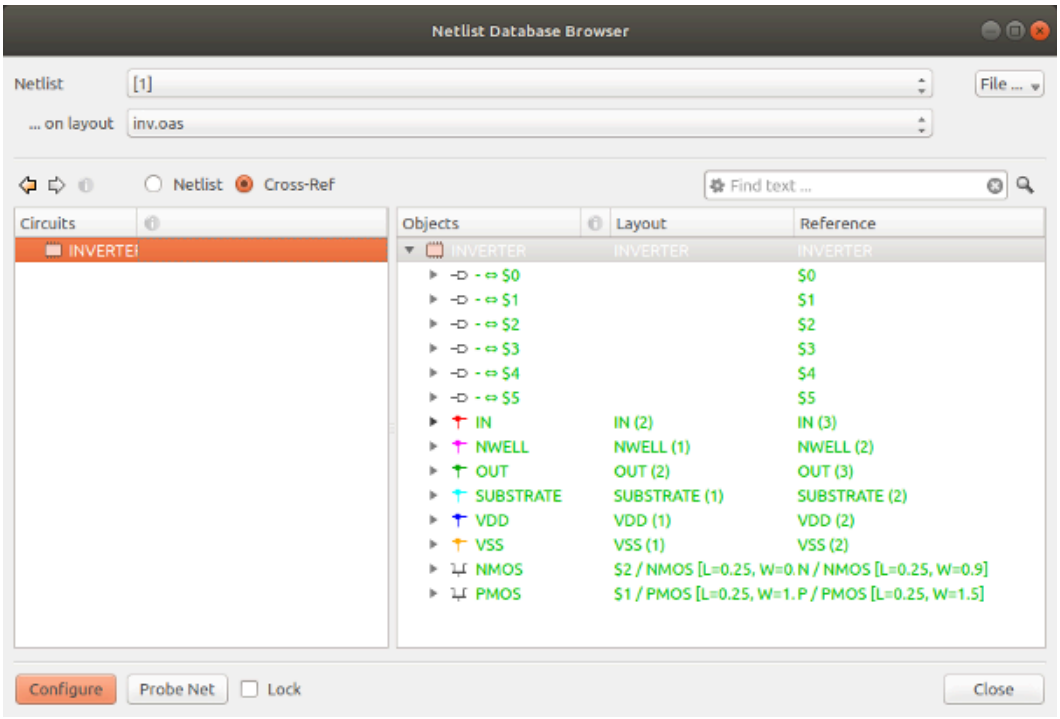
Sample LVS script

The LVS script to compare the layout above and the schematic now is this (for more details see LVS Reference):

```
# LVS script (demo technology, KLayout manual)

# Preamble:

deep

# Reports generated:

report_lvs     # LVS report window

# Drawing layers:

nwell       = input(1, 0)
```

```
active      = input(2, 0)
pplus       = input(3, 0)
nplus       = input(4, 0)
poly        = input(5, 0)
contact     = input(6, 0)
metal1      = input(7, 0)
metal1_lbl  = labels(7, 1)
via1        = input(8, 0)
metal2      = input(9, 0)
metal2_lbl  = labels(9, 1)

# Bulk layer for terminal provisioning:

bulk        = polygon_layer

# Computed layers:

active_in_nwell       = active & nwell
pactive               = active_in_nwell & pplus
pgate                 = pactive & poly
psd                   = pactive - pgate

active_outside_nwell  = active - nwell
nactive               = active_outside_nwell & nplus
ngate                 = nactive & poly
nsd                   = nactive - ngate

# Device extraction

# PMOS transistor device extraction
extract_devices(mos4("PMOS"), { "SD" => psd, "G" => pgate, "W" => nwell,
                                "tS" => psd, "tD" => psd, "tG" => poly, "tW" => nwell })

# NMOS transistor device extraction
extract_devices(mos4("NMOS"), { "SD" => nsd, "G" => ngate, "W" => bulk,
                                "tS" => nsd, "tD" => nsd, "tG" => poly, "tW" => bulk })

# Define connectivity for netlist extraction

# Inter-layer
connect(psd,        contact)
connect(nsd,        contact)
connect(poly,       contact)
connect(contact,    metal1)
connect(metal1,     metal1_lbl)   # attaches labels
connect(metal1,     via1)
connect(via1,       metal2)
connect(metal2,     metal2_lbl)   # attaches labels

# Global
connect_global(bulk,  "SUBSTRATE")
connect_global(nwell, "NWELL")

# Compare section

schematic("inv.cir")

align              # flattens unpaired circuits
netlist.simplify   # removes floating nets, combines devices

compare
```

For trying this script, load the inverter layout from "testdata/lvs/inv.oas" (KLayout sources) and open the Macro Editor IDE (Tools/Macro Development). Create a new script in the LVS tab and paste the text from above. Then run the script. The LVS report browser will open and show everything in green. This indicates the compare was successful:



Anatomy of the LVS script

The first and important statement of a LVS script should be the "deep" switch which enables hierarchical mode. Without hierarchical mode, the netlist is produced without subcircuits. Such flat netlist are inefficient to compare and hard to debug. Hence we switch to hierarchical mode with the "deep" statement (see deep):

```
deep
```

We also instruct LVS to create a report and open it in the report browser once LVS has finished:

```
report_lvs
```

We can also write the report to a file if we want (see report_lvs):

```
report_lvs("inv.lvsdb")
```

The next step is the declaration of the input layers:

```
nwell       = input(1, 0)
active      = input(2, 0)
pplus       = input(3, 0)
nplus       = input(4, 0)
poly        = input(5, 0)
contact     = input(6, 0)
metal1      = input(7, 0)
metal1_lbl  = labels(7, 1)
via1        = input(8, 0)
metal2      = input(9, 0)
metal2_lbl  = labels(9, 1)
```

"input" and "labels" are functions which pull layout layers from the layout source (the layout source is - as in DRC - usually the current layout). While "input" pulls all kind of shapes, "labels" will only pull texts. We use "labels" to pull labels for first metal from GDS layer 7, datatype 1 and labels for second metal from GDS layer 9, datatype 1. For details see input and labels.

In addition, we create an empty layer which we will need to represent the "substrate". This layer does not constitute a closed region but rather a heap of shapes which will all connect to the same (global) net later:

```
bulk = polygon_layer
```

The names we give to the layers are actually variables which represent a layout layer. As in DRC, we can use these to compute some derived layers:

```
active_in_nwell       = active & nwell
pactive               = active_in_nwell & pplus
pgate                 = pactive & poly
psd                   = pactive - pgate

active_outside_nwell  = active - nwell
nactive               = active_outside_nwell & nplus
ngate                 = nactive & poly
nsd                   = nactive - ngate
```

These formulas are all boolean operations. "&" is the boolean AND operation and "-" is the boolean NOT. Hence "active_in_nwell" is the part of "ACTIVE" which is inside "NWELL" while "active_outside_nwell" is the part of "ACTIVE" outside it. The main purpose of these formulas is to separate source and drain regions but cutting away the gate area from the "ACTIVE" area. This renders "psd" and "nsd" (PMOS and NMOS source/drain). The boolean operations are part of the DRC feature set. For more functions and detailed descriptions see DRC Reference: Layer Object.

We also separate gate regions for PMOS (pgate) and NMOS transistors (ngate) and with these ingredients we are ready to move to device extraction:

```
extract_devices(mos4("PMOS"), { "SD" => psd, "G" => pgate, "W" => nwell,
                                "tS" => psd, "tD" => psd, "tG" => poly, "tW" => nwell })
```

The first argument of "extract_devices" (see extract_devices) is the device extractor. The device extractor is an object responsible for the actual extraction of a certain device type. In our case the template is "MOS4" and we want to produce a new class of devices called "PMOS". `mos4("PMOS")` will create a new device extractor which produces devices of "MOS4" kind with class name "PMOS".

The second argument is a hash of layer symbols and layers. Each device extractor type defines a specific set of layer symbols. For all devices, two sets of the layers are required: the input layers which the extractor employs to recognize the device and the terminal connection layers which the extractor uses to place "magic" terminal shapes on. These polygons will create connections to the devices produced by the extractor.

The input layers are designated by upper-case letters, while the terminal output layers are designated with a lower-case "t" followed by the terminal name. The specification above is complete, but because "tW" defaults to "W" and "tS" and "tD" default to "SD", it can be written shorter as:

```
extract_devices(mos4("PMOS"), { "SD" => psd, "G" => pgate, "W" => nwell, "tG" => poly })
```

We also need an extractor for the "NMOS" class. It's built exactly the same way than the PMOS extractor:

```
extract_devices(mos4("NMOS"), { "SD" => nsd, "G" => ngate, "W" => bulk,
                                "tS" => nsd, "tD" => nsd, "tG" => poly, "tW" => bulk })
```

Having the devices is already half the work. We now need to supply the connectivity (see connect):

```
connect(psd,        contact)
connect(nsd,        contact)
connect(poly,       contact)
connect(contact,    metal1)
connect(metal1,     metal1_lbl)   # attaches labels
```

```
connect(metal1,     via1)
connect(via1,       metal2)
connect(metal2,     metal2_lbl)   # attaches labels
```

These statements will connect PMOS source/drain regions (psd) with CONTACT regions (contact), NMOS source/drain regions (nsd) also with CONTACT. POLY will also connect to CONTACT. Remember that we specified psd, nsd and poly as terminal outputs "tS", "TD" and "tG" in the device extraction. By including these layers into the connectivity, we establish device terminal connections to the nets formed by these layers.

The metal stack is trivial (CONTACT to METAL1, METAL1 to METAL2 via VIA1). The labels are attached to nets simply by including the label layers into the connectivity. The net extractor will pull the text strings from these connected text objects and assign them to the nets as net names.

Furthermore, two special connections need to be made (see connect_global):

```
connect_global(bulk,  "SUBSTRATE")
connect_global(nwell, "NWELL")
```

Global connections basically say that all shapes on a certain layer belong to the same net - even if they do not touch - and this net is always shared between circuits and subcircuits. This is certainly true for the bulk layer, but not necessarily for the NWELL layer. Isolated NWELL patches do not connect together. We will correct this small error later when it comes to extraction with tie-down diodes.

We have now provided all the essential inputs for the netlist formation. We now have to specify the reference netlist:

```
schematic("inv.cir")
```

Two optional, but recommended steps are hierarchy alignment and extracted netlist simplification:

```
align              # flattens unpaired circuits
netlist.simplify   # removes floating nets, combines devices
```

"align" will remove circuits which are not present in the other netlist by integrating their content into the parent cell. This will remove auxiliary cells which are usually present in a layout but don't map to a schematic cell (e.g. device PCells). "netlist.simplify" reduces the netlist by floating nets, performs device combination (e.g. fingered transistors). This method will also create pins from labeled nets in the top level circuit.

The order should be "align", then "netlist.simplify". Both have to happen before "compare" to be effective. "align" is described in LVS Compare, "netlist.simplify" in LVS Netlist Tweaks.

Finally after having set this up, we can trigger the compare step:

```
compare
```

If we insert a netlist write statement (see target_netlist) at the beginning of the script, we can obtain a SPICE version of the extracted netlist:

```
# SPICE output statement (insert at beginning of script):
target_netlist("inv_extracted.cir", write_spice, "Extracted by KLayout")
```

Since we have a LVS match, the extracted netlist is pretty much the same than the reference netlist, but enhanced by some geometrical parameters such as source and drain area and perimeter:

```
* Extracted by KLayout

* cell INVERTER
.SUBCKT INVERTER
* net 1 IN
* net 2 VSS
* net 3 VDD
* net 4 OUT
```

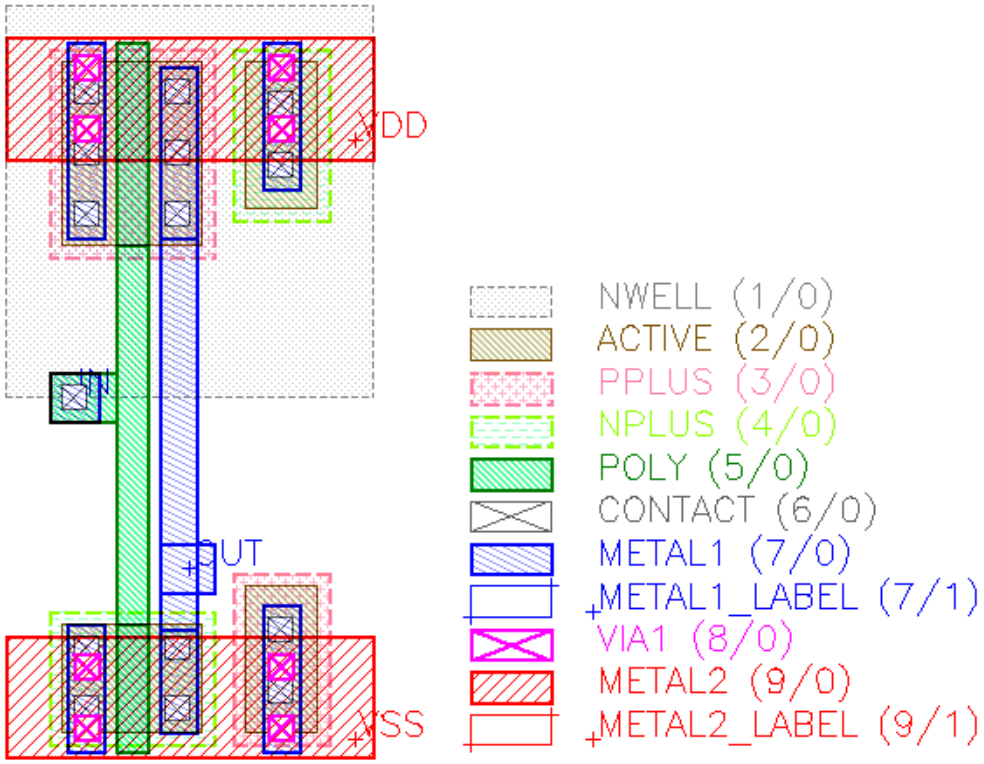
```
* net 5 NWELL
* net 6 SUBSTRATE
* device instance $1 r0 *1 1.025,4.95 PMOS
M$1 3 1 4 5 PMOS L=0.25U W=1.5U AS=0.675P AD=0.675P PS=3.9U PD=3.9U
* device instance $2 r0 *1 1.025,0.65 NMOS
M$2 2 1 4 6 NMOS L=0.25U W=0.9U AS=0.405P AD=0.405P PS=2.7U PD=2.7U
.ENDS INVERTER
```

Inverter with tie-down diodes

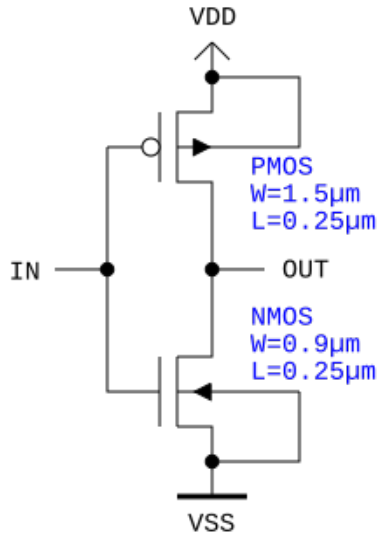
The inverter cell above is not useful by itself as it lacks features to tie the n well and the substrate to a defined potential. This is achieved with tie-down diodes.

Tie-down diodes are contacts over active regions. The active regions are implanted p+ on the substrate and n+ within the n well (the opposite implant type of transistors). With this doping profile, the metal contact won't form a Schottky barrier to the Silicon bulk and behave like an ohmic contact. So in fact, the "diode" isn't a real diode in the sense of a rectifier.

The modified layout is this one:



The corresponding schematic is this:



Inverter with
tie-down diodes

With this circuit, the n well is always at VDD potential and the substrate is tied at VSS:

```
* Simple CMOS inverer circuit
.SUBCKT INVERTER_WITH_DIODES VSS IN OUT VDD
Mp VDD IN OUT VDD PMOS W=1.5U L=0.25U
Mn OUT IN VSS VSS NMOS W=0.9U L=0.25U
.ENDS
```

The LVS script is slightly longer when extraction of tie-down diodes is included:

```
# LVS script (demo technology, KLayout manual)

# Preamble:

deep

# Reports generated:

report_lvs     # LVS report window

# Drawing layers:

nwell       = input(1, 0)
active      = input(2, 0)
pplus       = input(3, 0)
nplus       = input(4, 0)
poly        = input(5, 0)
contact     = input(6, 0)
metal1      = input(7, 0)
metal1_lbl  = labels(7, 1)
```

```
via1        = input(8, 0)
metal2      = input(9, 0)
metal2_lbl  = labels(9, 1)

# Bulk layer for terminal provisioning

bulk        = polygon_layer

# Computed layers

active_in_nwell       = active & nwell
pactive               = active_in_nwell & pplus
pgate                 = pactive & poly
psd                   = pactive - pgate
ntie                  = active_in_nwell & nplus

active_outside_nwell  = active - nwell
nactive               = active_outside_nwell & nplus
ngate                 = nactive & poly
nsd                   = nactive - ngate
ptie                  = active_outside_nwell & pplus

# Device extraction

# PMOS transistor device extraction
extract_devices(mos4("PMOS"), { "SD" => psd, "G" => pgate, "W" => nwell,
                                "tS" => psd, "tD" => psd, "tG" => poly, "tW" => nwell })

# NMOS transistor device extraction
extract_devices(mos4("NMOS"), { "SD" => nsd, "G" => ngate, "W" => bulk,
                                "tS" => nsd, "tD" => nsd, "tG" => poly, "tW" => bulk })

# Define connectivity for netlist extraction

# Inter-layer
connect(psd,        contact)
connect(nsd,        contact)
connect(poly,       contact)
connect(ntie,       contact)
connect(nwell,      ntie)
connect(ptie,       contact)
connect(contact,    metal1)
connect(metal1,     metal1_lbl)   # attaches labels
connect(metal1,     via1)
connect(via1,       metal2)
connect(metal2,     metal2_lbl)   # attaches labels

# Global
connect_global(bulk, "SUBSTRATE")
connect_global(ptie, "SUBSTRATE")

# Compare section

schematic("inv2.cir")

align
netlist.simplify

compare
```

The main difference is the computation of the regions for n tie-down (inside n well) and p tie-down. This is pretty straightforward:

```
ntie                  = active_in_nwell & nplus
ptie                  = active_outside_nwell & pplus
```

Device extraction does not change, but we need to include the tie-down regions into the connectivity:

```
connect(ntie,       contact)
connect(nwell,      ntie)
connect(ptie,       contact)
```

By connecting ntie to contact and nwell, we readily establish a connection to n well which behaves then like a conductive layer (although the resistance will be very high). Remember the the device extractors for PMOS will put the bulk terminals on nwell too, so the transistor is automatically connected to the nwell net.

ptie cannot be simply connected as there are no polygons for "substrate". But we can include ptie in the global connections:

```
connect_global(bulk, "SUBSTRATE")
connect_global(ptie, "SUBSTRATE")
```

nwell is no longer included in the global connections, hence we do no longer and incorrectly consider all nwell regions to be connected.

The extracted netlist shows the bulk terminals of NMOS and PMOS connected to source (drain and source are equivalent):

```
* Extracted by KLayout

* cell INVERTER_WITH_DIODES
.SUBCKT INVERTER_WITH_DIODES
* net 1 IN
* net 2 VDD
* net 3 OUT
* net 4 VSS
* device instance $1 r0 *1 1.025,4.95 PMOS
M$1 2 1 3 2 PMOS L=0.25U W=1.5U AS=0.675P AD=0.675P PS=3.9U PD=3.9U
* device instance $2 r0 *1 1.025,0.65 NMOS
M$2 4 1 3 4 NMOS L=0.25U W=0.9U AS=0.405P AD=0.405P PS=2.7U PD=2.7U
.ENDS INVERTER_WITH_DIODES
```

1.5.3. LVS Devices

Device extractors and device classes

KLayout provides two concepts for handling device variety:

Device classes are device categories. There are general categories such as resistors or MOS transistors. Specific categories can be created to represent specific incarnations - e.g. NMOS and PMOS devices. Device classes also determine how devices combine.

Device classes are documented here: LVS Device Classes.

Device extractors are the actual worker objects that analyze layout and produce devices. As for device classes, there are general device extractors. Each device extractor produces devices from a specific class.

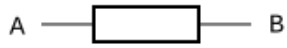
Device extractors are documented here: LVS Device Extractors.

1.5.4. LVS Device Classes

KLayout implements a variety of standard device classes. These device classes are the basis for forming particular incarnations of device classes. For example, the MOS4 class is the basis for the specific device classes for NMOS and PMOS transistors.

Resistor

DeviceClassResistor



The plain resistor has two terminals, A and B. It features the following parameters:

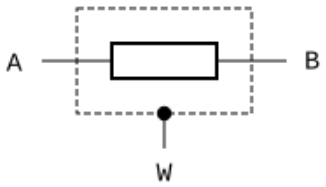
- **R** : The resistance value in Ohm
- **L** : The length in µm
- **W** : The width in µm
- **A** : The area of the resistor area in µm²
- **P** : The perimeter of the resistor area in µm

Resistors can combine in parallel or serial fashion.

In SPICE, plain resistors are represented by the "R" element. The API class is DeviceClassResistor.

Resistor with bulk terminal

```
DeviceClassResistorWithBulk
```



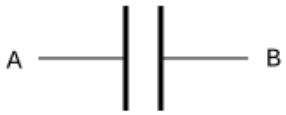
The resistor with bulk terminal is an extension of the plain resistor. If has the same parameters, but one additional terminal (W) which connects to the area the resistor sits in (e.g. well or substrate).

Resistors with bulk can combine in parallel or serial fashion if their bulk terminals are connected to the same net.

The API class of the resistor with bulk is DeviceClassResistorWithBulk.

Capacitor

```
DeviceClassCapacitor
```



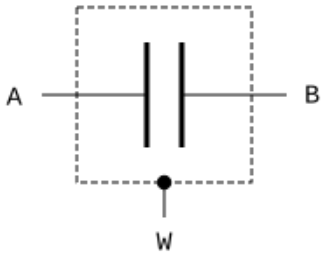
The plain capacitor has two terminals, A and B. It features the following parameters:

- **C** : The capacitance value in Farad
- **A** : The area of the capacitor area in µm²
- **P** : The perimeter of the capacitor area in µm

In SPICE, plain capacitors are represented by the "C" element. The API class is DeviceClassCapacitor.

Capacitor with bulk terminal

DeviceClassCapacitorWithBulk



The capacitor with bulk terminal is an extension of the plain capacitor. If has the same parameters, but one additional terminal (W) which connects to the area the capacitor sits in (e.g. well or substrate).

Capacitors with bulk can combine in parallel or serial fashion if their bulk terminals are connected to the same net.

The API class of the capacitor with bulk is DeviceClassCapacitorWithBulk.

Diode

DeviceClassDiode

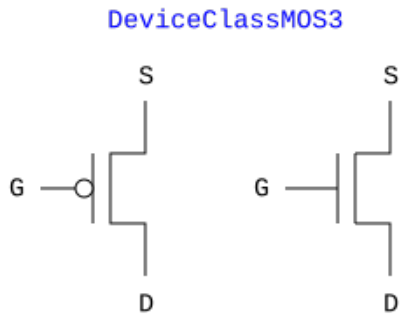
Diodes have two terminals, A and C for anode and cathode. Diodes feature the following parameters:

- **A** : The area of the diode in µm²
- **P** : The perimeter of the diode in µm

Diodes combine in parallel (A to A and C to C). In this case their areas and perimeters will add.

In SPICE, diodes are represented by the "D" element using the device class name as the model name. The API class is DeviceClassDiode.

MOS transistor



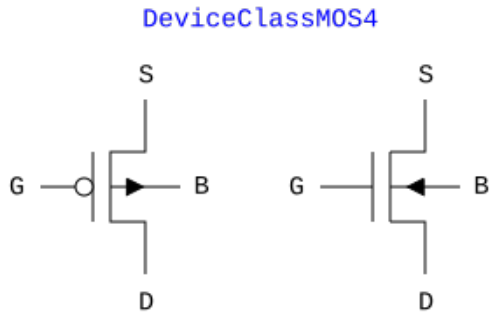
Three-terminal MOS transistors have terminals S, G and D for source, gate and drain. S and D are commutable. They feature the following parameters:

- **W** : The gate width in µm
- **L** : The gate (channel) length in µm
- **AS** : The source area in µm²
- **PS** : The source perimeter in µm
- **AD** : The drain area in µm²
- **PD** : The drain perimeter in µm

MOS3 transistors combine in parallel when the source/drains and gates are connected and their gate lengths are identical. In this case their widths, areas and perimeters will add.

MOS transistor with bulk

The API class of the three-terminal MOS transistor is DeviceClassMOS3Transistor.

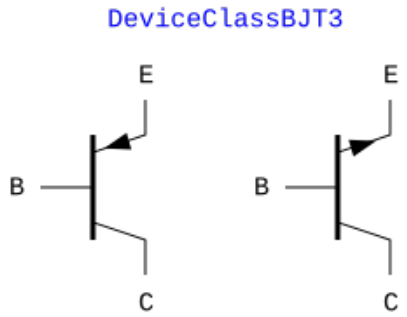


The four-terminal transistor is an extension of the three-terminal one and offers an additional bulk terminal (B). It is probably the most prominent transistor device as the four-terminal version is compatible with the SPICE "M" element.

MOS transistors with bulk can combine in parallel the same way the three-terminal versions do if their bulk terminals are connected to the same net.

In SPICE, MOS4 devices are represented by the "M" element with the device class name as the model name. The API class is DeviceClassDiode.

Bipolar transistor



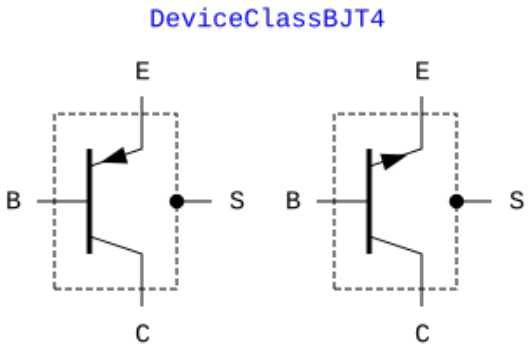
The three-terminal bipolar transistor can be either NPN or PNP type. In KLayout, this device type can represent both lateral and vertical types. The parameters are:

- **AE** : The emitter area in µm²
- **PE** : The emitter perimeter in µm
- **NE** : The emitter count (initially 1)
- **AB** : The base area in µm²
- **PB** : The base perimeter in µm
- **AC** : The collector area in µm²
- **PC** : The collector perimeter in µm

Upon extraction, multi-emitter versions are extracted as multiple devices - one for each emitter area - and NE = 1. Bipolar transistors combine when in parallel. In this case, their emitter parameters AE, PE and NE are added.

In SPICE, BJT3 devices are represented by the "Q" element with the device class name as the model name. The API class is DeviceClassBJT3Transistor.

Bipolar transistor with substrate



The four-terminal transistor is an extension of the three-terminal one and offers an additional bulk terminal (S).

Bipolar transistors with bulk can combine in parallel the same way the three-terminal versions do if their bulk terminals are connected to the same net.

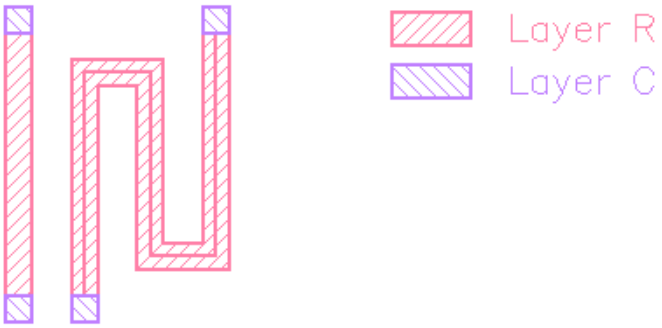
In SPICE, BJT4 devices are represented by the "Q" element with four nodes and the device class name as the model name. The API class is DeviceClassBJT4Transistor.

1.5.5. LVS Device Extractors

Device extractors and the actual "workers" of the device extraction process. KLayout comes with a variety of pre-built device extractors. It's possible to implement custom device extractors in the framework of LVS scripts (speaking Ruby).

Resistor extractors (resistor and resistor_with_bulk)

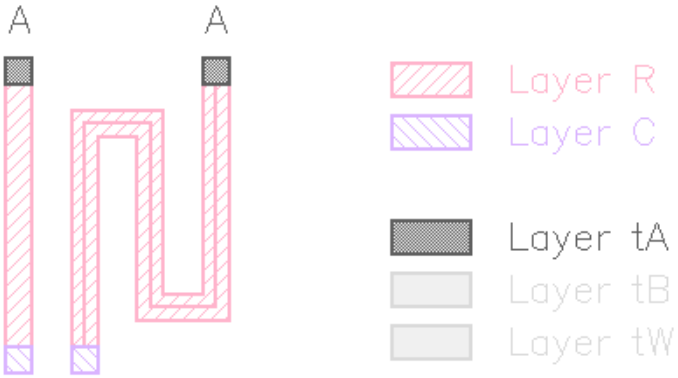
The resistor extractor assumes a layout which consists of a resistor "wire" and two caps (contacts). The wire is specified with the layer symbol "R", the caps are specified with the layer symbol "C":

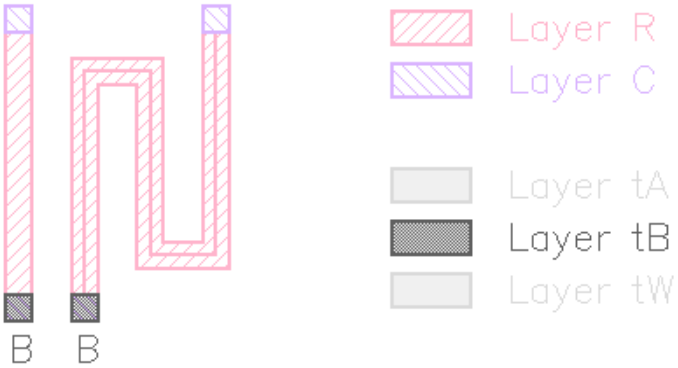


The extractor will compute the resistance from the number of squares and the sheet resistance. The sheet resistance needs to be given when creating the extractor:

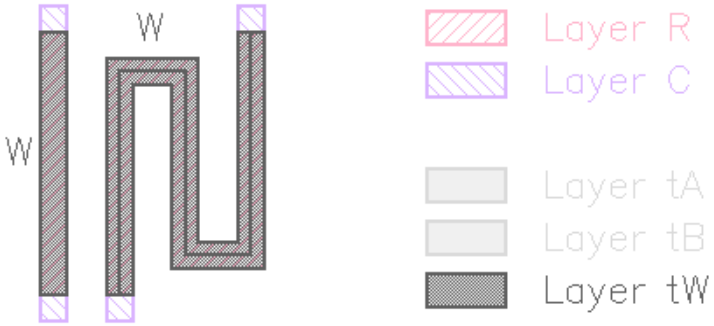
```
sheet_rho = 0.5
model_name = "RES"
extract_devices(resistor(model_name, sheet_rho), { "R" => res_layer, "C" => contact_layer })
```

The plain resistor offers two terminals which it outputs on "tA" and "tB" terminal layers. If "tA" or "tB" is not specified, "A" or "B" terminals will be written on the "C" layer. respectively.





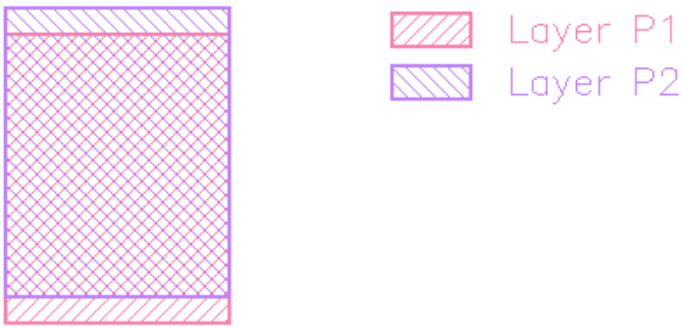
For the resistor with bulk, the wire area is output on the "tW" terminal layer as the "W" terminal:



Note: The resistance computation is based on a simple approximation. It computes the number of squares by tracing the perimeter of the "R" polygon. The perimeter length is separated in parts where the perimeter touches the "C" layer and parts where it does not. The number of squares is computed from the non-touching length divided by the touching length.

Capacitor extractors (capacitor and capacitor_with_bulk)

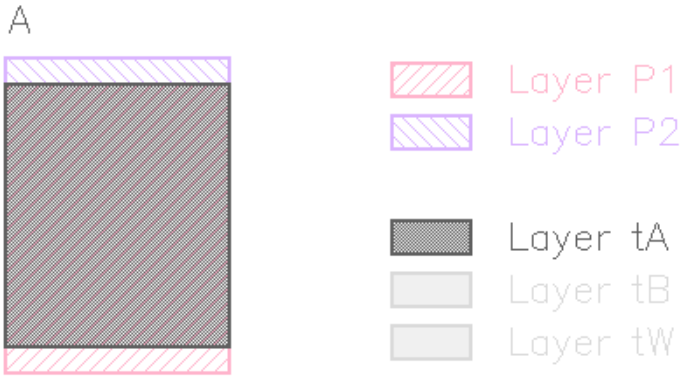
Capacitors are assumed to consist of two "plates" (vertical capacitors). The plates are on layers P1 and P2. The capacitor is extracted from the area where these two layers overlap.

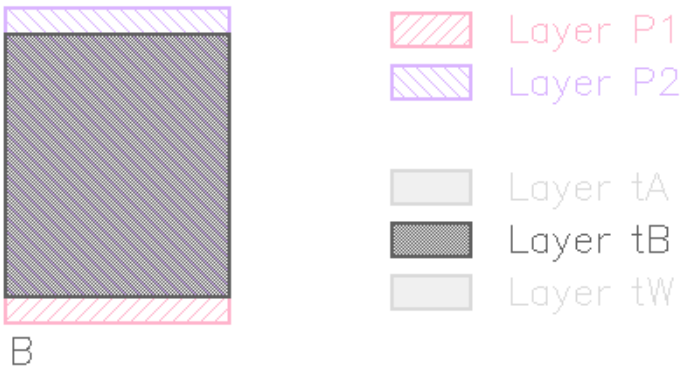


The extractor will compute the capacitance from the area of the overlap and the capacitance per area (F/µm²) value.

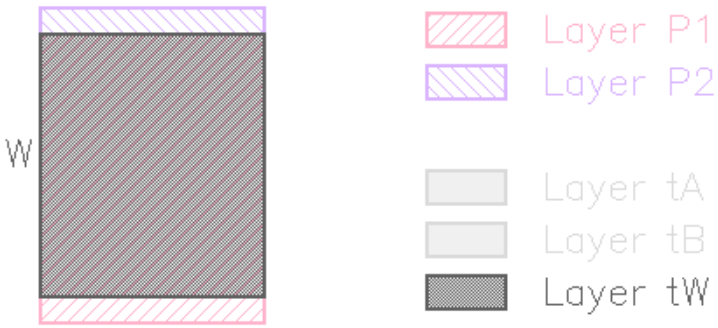
```
area_cap = 1.5e-15
model_name = "CAP"
extract_devices(capacitor(model_name, area_cap), { "P1" => metal1, "P2" => metal2 })
```

The plain capacitor offers two terminals which it outputs on "tA" and "tB" terminal layers. If "tA" or "tB" is not specified, "A" or "B" terminals will be written on the "P1" and "P2" layers respectively.



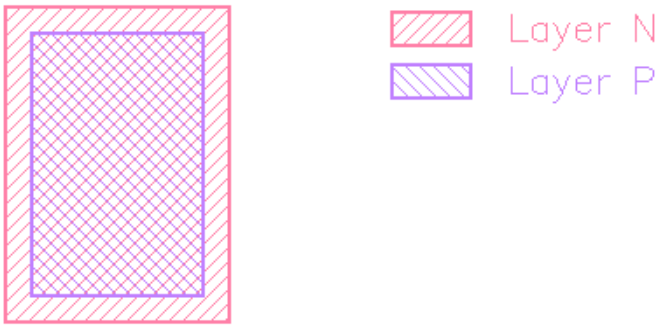


For the capacitor with bulk, the capacitor area is output on the "tW" terminal layer as the "W" terminal:



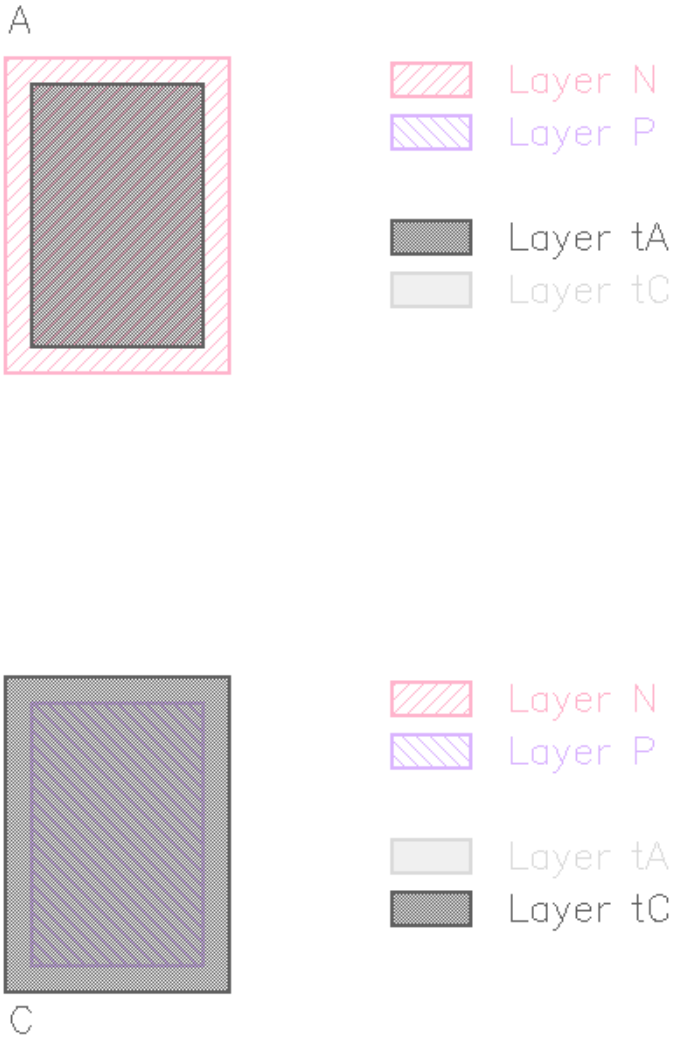
Diode extractor (diode)

Diodes are assumed to consist of two vertical implant regions (wells, diffusion). One of the regions is p type ("P" layer) and the other "n" type ("N" layer). These layers also form the anode (p) and cathode (n) of the diode.



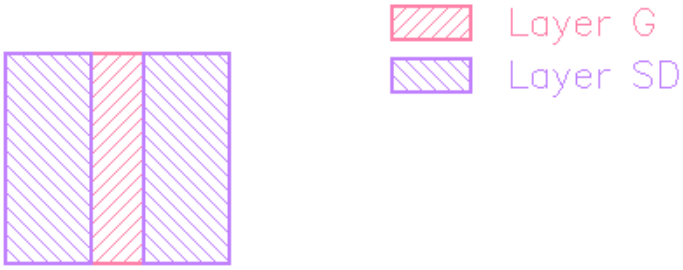

```
model_name = "DIODE"
extract_devices(diode(model_name), { "P" => pplus, "N" => nwell })
```

The diode offers two terminals which it outputs on "tA" and "tC" terminal layers. If "tA" is not specified, "A" terminals will be written on the "P" layer. If "tC" is not specified, "C" terminals will be written on the "N" layer.



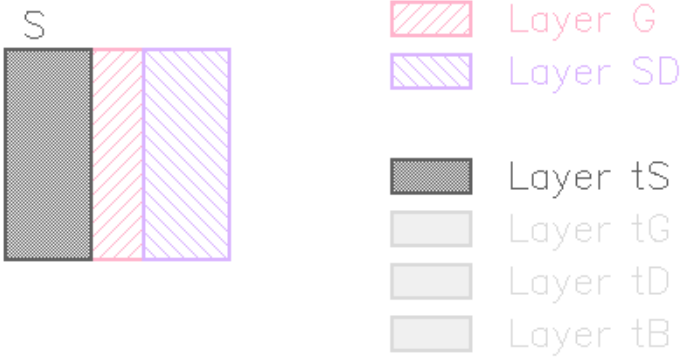
MOS transistor extractor (mos3 and mos4)

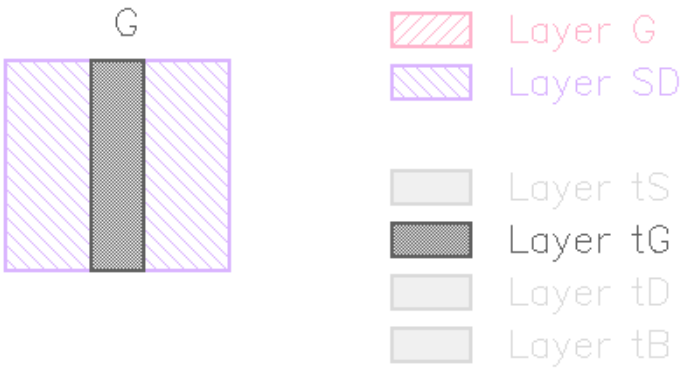
MOS transistors are recognized by their gate ("G" input) and source/drain ("SD" input) regions. Source and drain needs to be separated from the gate shape. The touching edges of gate and source/drain regions define the width of the device, the perpendicular dimension the gate length. Because the separation of source/drain, the computation of gates and the separation of these for NMOS and PMOS devices, the "G" and "SD" layers are usually derived layers. As these usually won't participate in the connectivity, it's important to specify the "tS", "tD", "tG" and "tB" (for MOS4) layers explicitly and redirect the terminal shapes to layers that really participate in connections.

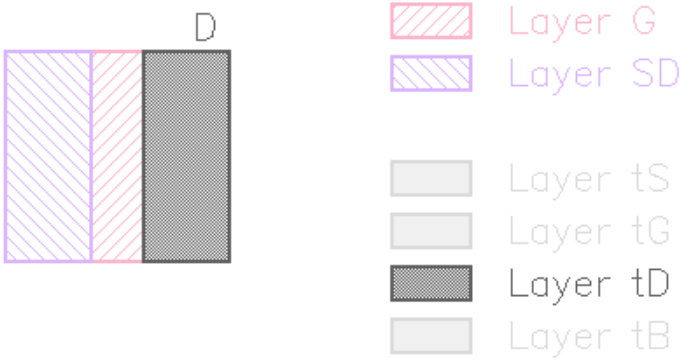



```
model_name = "PMOS"
extract_devices(mos4(model_name), { "SD" => (active - poly) & pplus, "G" => (active & poly), "W" => nwell,
                                    "tS" => active, "tD" => active, "tG" => poly, "tB" => nwell })
```

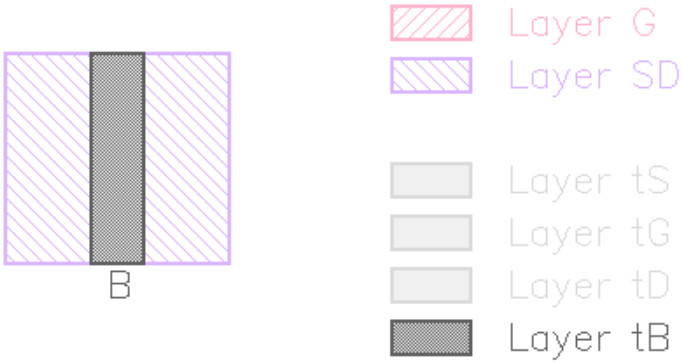
The MOS3 device produces three terminals which it outputs on "tS", "tG" and "tD" terminal layers (source, gate and drain respectively):







The MOS4 device offers one more terminal (bulk) which it writes on "tB".



Diffusion MOS transistor extractor (dmos3 and dmos4)

DMOS devices are basically identical to MOS devices, but for those source and drain are separated. This is often the case for diffusion MOS transistory, hence this name.

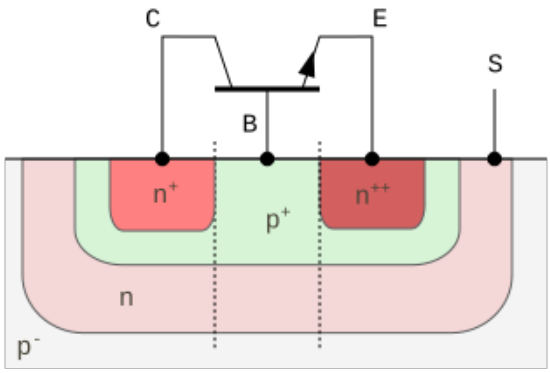
DMOS and MOS devices share the same device class. DMOS devices are configured such that source and drain cannot be swapped. The netlist compare will report source/drain swapping as errors for such devices.

DMOS transistors are recognized by their gate ("G" input), source ("S" input) and drain ("D" input) regions. Source and drain needs to be separated from the gate shape. The touching edges of gate and source/drain regions define the width of the device, the perpendicular dimension the gate length. The terminal output layers for DMOS devices are the same than for MOS devices: "tS" for source, "tD" for drain, "tG" for gate, "tB" for bulk (4-terminal version).

Bipolar transistor extractor (bjt3 and bjt4)

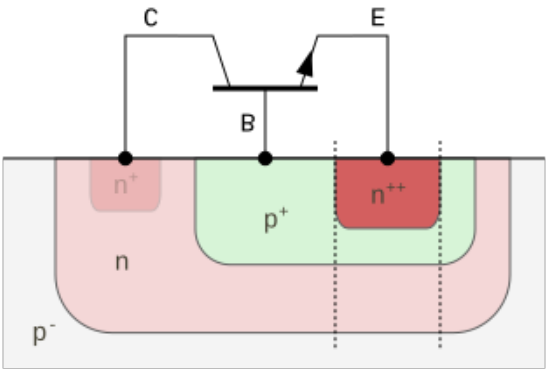
There are basically two kind of bipolar transistors: vertical and lateral ones.

Lateral transistors are formed by implant or diffusion wells creating a intermittent n/p structure on the wafer. The basic recognition region is the base region. The collector and emitter regions are inside or overlapping the base region and use the opposite doping than base: if the base region is n doped, then collector and emitter regions have to be p doped. The structure then forms a PNP transistor. KLayout recognizes lateral transistors when the base is **partially** covered by the collector region. For lateral transistors, the emitter is defined by the emitter region inside base. The collector region is defined by collector inside base and outside emitter.



(lateral NPN transistor)

Vertical transistors are formed by a stack of n/p wells. Sometimes vertical transistors are formed as parasitic devices in standard CMOS processes. A PNP transistor can be formed by taking the collector as the substrate, nwell for the base and pplus implant for the emitter. KLayout recognizes a vertical bipolar transistor when the base is covered **entirely** by the collector or has **no collector at all** - this means the collector region can be empty (e.g. bulk).

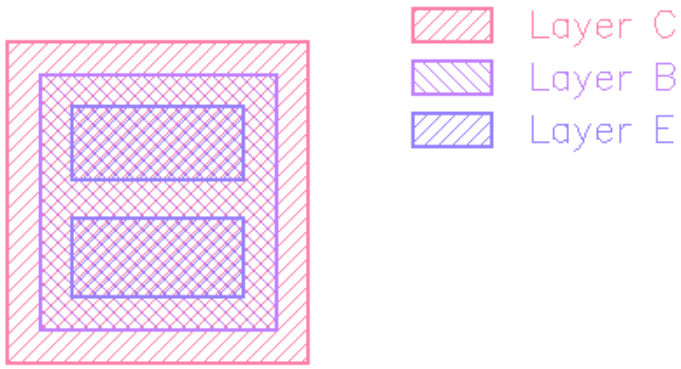


(vertical NPN transistor)

In both cases, there can be multiple emitter regions inside a base island. In this case, one transistor is extracted for each emitter region.

Vertical bipolar transistors

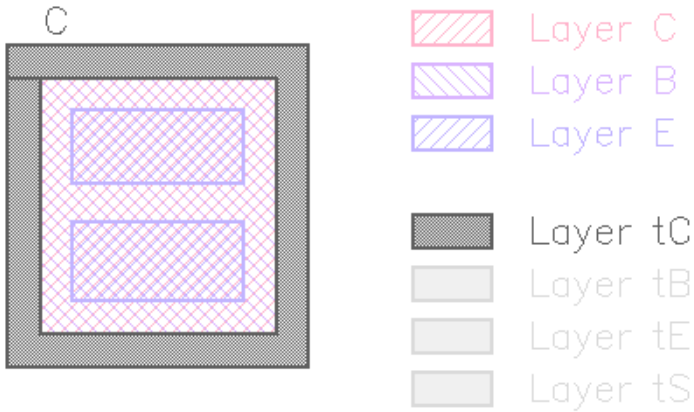
Vertical bipolar transistors take their inputs from "B" (base), "C" (collector) and "E" (emitter). "C" is optional:

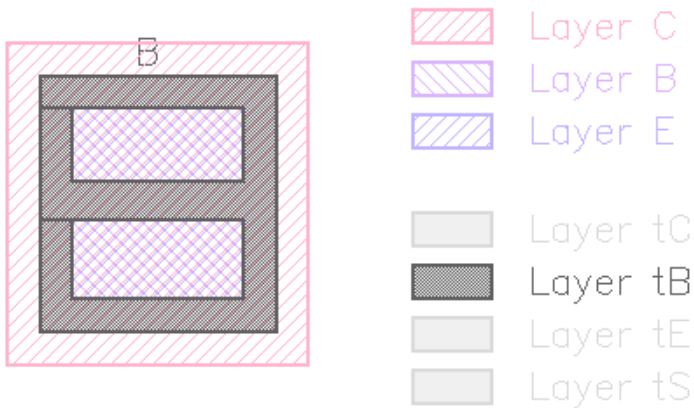


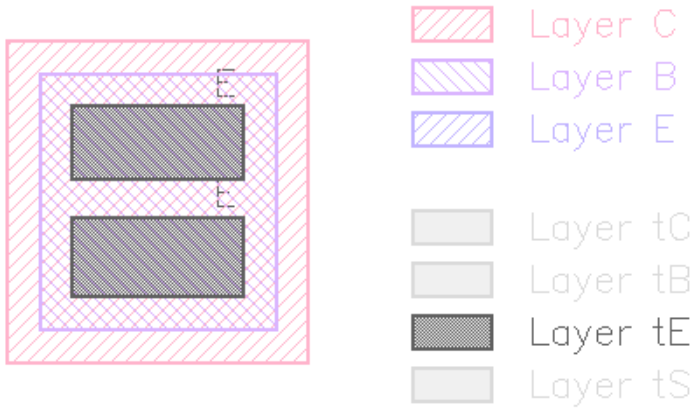
Especially for bipolar devices it's important to device useful terminal output layers. Typically, the wells and diffusion areas will be connected through "contact", (not considering the Schottky diodes for now). So it's a good idea to send the terminals to the contact layer:

```
model_name = "PNP"
extract_devices(bjt3(model_name), { "C" => collector, "B" => base, "E" => emitter,
                                    "tC" => contact, "tB" => contact, "tE" => contact })
```

The BJT3 device produces three terminals which it outputs on "tC", "tB" and "tE" terminal layers (collector, base and emitter respectively):

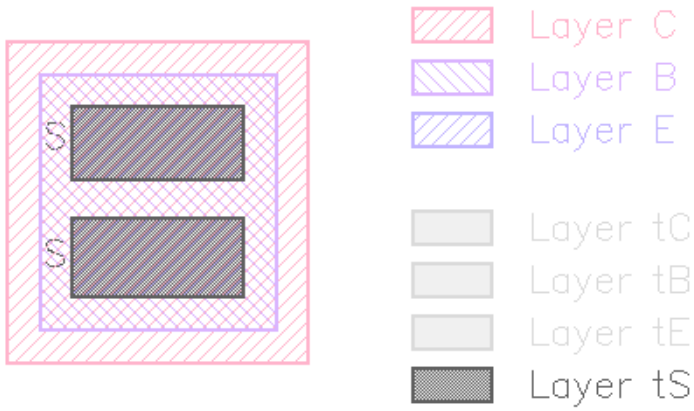

C
Layer C
Layer B
Layer E
Layer tC
Layer tB
Layer tE
Layer tS


B
Layer C
Layer B
Layer E
Layer tC
Layer tB
Layer tE
Layer tS



If the collector region is empty (e.g. p substrate), the base shape is copied to the "tC" output layer for the collector terminal.

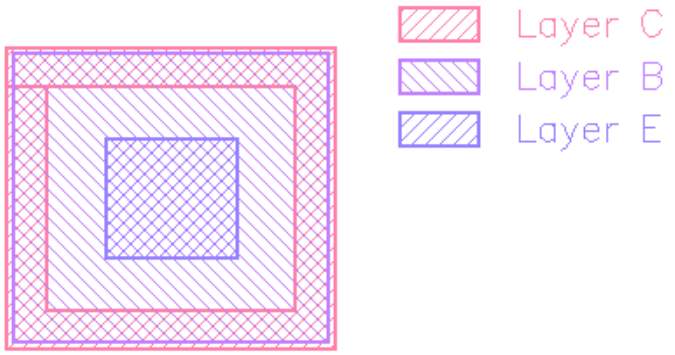
The BJT4 device offers one more terminal (substrate) which it writes on "tS". "tS" is a copy of the emitter shape but connected to the substrate terminal:



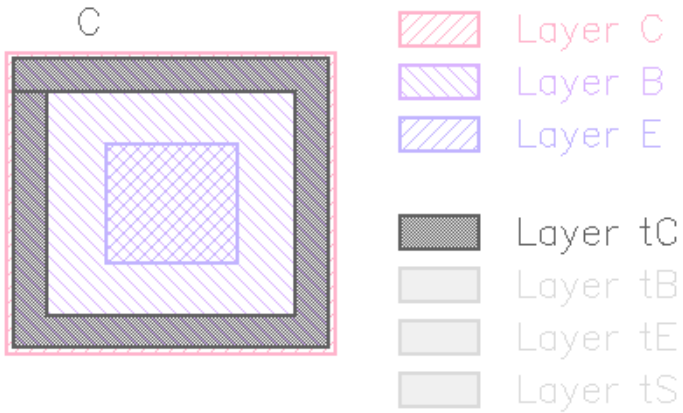
Lateral bipolar transistors

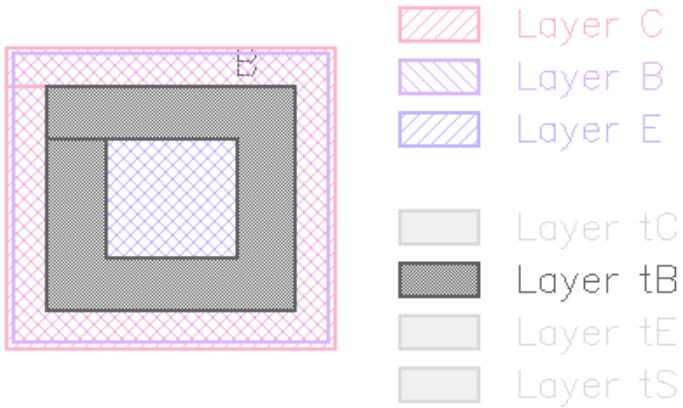
Lateral bipolar transistors also take their inputs from "B" (base), "C" (collector) and "E" (emitter). For lateral transistors, "C" is not optional and must not fully cover the base region. Apart from this, the use model for BJT3 and BJT4 extractors is identical for vertical and lateral transistors.

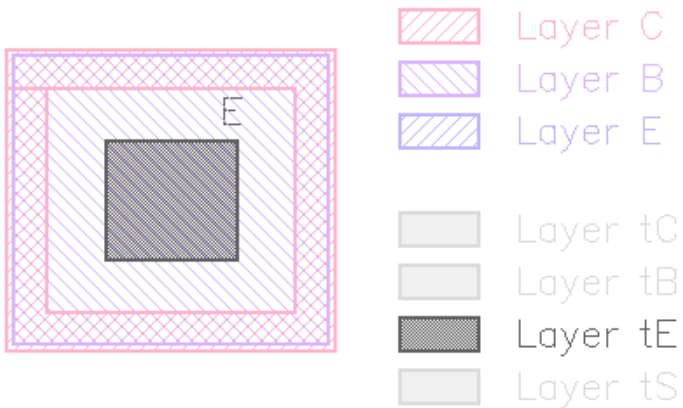
A typical lateral transistor is formed by a collector ring and emitter island inside the base region:



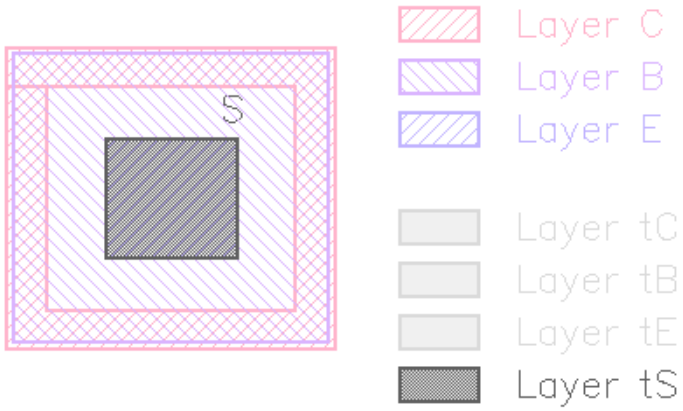
The terminals produced by the bipolar transistor extractor in the lateral case are the same than for the vertical case, but with a different geometry:







Again, for BJT4, "tS" is a copy of the emitter shape but connected to the substrate terminal:



Device extractors and device classes

"extract_devices" will return the DeviceClass object of the devices generated. This object can be useful to apply some basic modifications. The most important of them is enabling or disabling certain parameters.

Most device extractors extract more parameters than they give you by default. For example, the resistor extractor will not just extract the resistance, but also the length (L) and width (W) of the resistor stripe and its area (A) and perimeter (P). By default these additional parameters are declared "secondary" - i.e. they will not participate in the device compare and will not be netlisted.

Parameters can be fully enabled by using enable_parameter or disabled using disable_parameter. tolerance can be used to enable a parameter for compare and to specify a compare tolerance. ignore_parameter can be used to ignore a parameter in the compare step.

Another way of customizing the built-in device extractors is to supply a custom device class. The following code creates a new resistor class which changes the preconfigured device parameter definitions to enable "W" and "L".

```
class MyResistor < RBA::DeviceClassResistor
  def initialize
    super
    enable_parameter("W", true)
    enable_parameter("L", true)
  end
end

...

extract_devices(resistor("RES", 1, MyResistor), ...)
```

Using a custom device class opens the option to supply additional parameters for example or to implement some entirely new device while using the extraction mechanics of the resistor extractor. The only requirement is compatibility of the parameter and terminal definitions.

1.5.6. LVS Input/Output

LVS (and also DRC as far as netlist extraction is concerned) provides interfaces to write and read netlists/schematics, annotated layout and LVS results. There are three major categories of I/O:

- **Netlist**: this is the plain circuit information. With subcircuit this forms a hierarchical netlist. Currently, the format available to import and export netlists is a certain SPICE netlist flavor. It's possible to customize the reading and writing process to achieve some flexibility.
- **Layout-to-netlist database (L2N DB)**: also called extracted netlist or annotated layout. This is the netlist taken from the original layout together with the corresponding shapes. This database allows reconstructing a net geometrically as far as non-device shapes are involved. Devices are abstracted by their terminal geometries.
- **LVS result database (LVS DB)**: this is the L2N database plus the reference netlist and a "cross reference": a list of paired circuits, nets, devices, pins and subcircuits and status information. The cross-reference is both a lookup table and a debugging aid.

Writing netlists

You can write a netlist file to supply netlists for (functional) simulators for example. Within LVS scripts, the global "target_netlist" statement triggers writing of a netlist (see target_netlist for details).

```
target_netlist("output.cir", write_spice, "Created by KLayout")
```

This statement can basically appear anywhere in the LVS script. The netlist will written after the script has executed successfully. The first argument is the file's path (by default relative to the original layout file). The second argument is the "writer". "write_spice" creates a netlist writer writing SPICE format with a limited degree of flexbility. See below for customizing the writer. The third argument finally is an (optional) comment which will be written into the netlist as a header.

The "write_spice" configuration function has two options:

```
write_spice(use_net_names, with_comments)
```

Both options are boolean values. If true and present, the first option will make the writer use the real net's names instead of numerical IDs. If true and present, "with_comments" will embed debug comments into the netlist showing instance locations, pin names etc.

Further customization can be achieved by providing an explicit SPICE writer with a delegate (see NetlistSpiceWriterDelegate). For doing so, subclass NetlistSpiceWriterDelegate and reimplement one or several of the methods provided for reimplementation. Those are NetlistSpiceWriterDelegate#write_device, NetlistSpiceWriterDelegate#write_device_intro and NetlistSpiceWriterDelegate#write_header.

Here is an example that supplied subcircuit models rather than device elements:

```
# Write extracted netlist to extracted.cir using a special
# writer delegate

# This delegate makes the writer emit subcicuit calls instead of
# standard elements for the devices
class SubcircuitModels < RBA::NetlistSpiceWriterDelegate

  def write_header
    emit_line(".INCLUDE 'models.cir'")
  end

  def write_device(device)
    str = "X" + device.expanded_name
    device_class = device.device_class
    device_class.terminal_definitions.each do |td|
      str += " " + net_to_string(device.net_for_terminal(td.id))
    end
    str += " " + device_class.name
    str += " PARAMS:"
    device_class.parameter_definitions.each do |pd|
      str += " " + pd.name + ("=%.12g" % device.parameter(pd.id))
```

```
    end
    emit_line(str)
  end

end

# Prepare a writer using the new delegate
custom_spice_writer = RBA::NetlistSpiceWriter::new(SubcircuitModels::new)
custom_spice_writer.use_net_names= true
custom_spice_writer.with_comments = false

# The declaration of netlist production using the new custom writer
target_netlist("extracted.cir", custom_spice_writer, "Extracted by KLayout")
```

This script will produce the following netlist for the simple inverter from the LVS introduction. Instead of printing "M" elements - which is the default - subcircuit calls are produced. This allows putting more elaborate models into subcircuits. The device class name addresses these model subcircuits:

```
* Extracted by KLayout
.INCLUDE 'models.cir'

.SUBCKT INVERTER
X$1 VDD IN OUT NWELL PMOS PARAMS: L=0.25 W=1.5 AS=0.675 AD=0.675 PS=3.9 PD=3.9
X$2 VSS IN OUT SUBSTRATE NMOS PARAMS: L=0.25 W=0.9 AS=0.405 AD=0.405 PS=2.7
+ PD=2.7
.ENDS INVERTER
```

Netlists can be written directly from the netlist object. Within the script, the netlist object can be obtained with the netlist function. This function will first trigger a netlist extraction unless this was done already and return a Netlist object. Use Netlist#write to write this netlist object then. Unlike "target_netlist", this method is executed immediately and this way, a single netlist can be written to multiple files in different flavours.

Reading netlists

The main use case for reading netlists is for comparison in LVS. Reference netlists are read with the "schematic" function (see schematic):

```
schematic("inverter.cir")
```

Currently SPICE is understood with some limitations: Only a subset of elements is implemented by default. These are "M" (gives "MOS4" device classes), "Q" (gives BJT3 or BJT4 device classes), "R" (gives Resistor device classes), "C" (gives Capacitor device classes) and "D" (gives diode device classes).

As for the SPICE reader, a delegate can be provided to customize the reader. For doing so, subclass the NetlistSpiceReaderDelegate class and reimplement the methods provided. These are: NetlistSpiceReaderDelegate#wants_subcircuit, NetlistSpiceReaderDelegate#element, NetlistSpiceReaderDelegate#finish and NetlistSpiceReaderDelegate#start

This example customizes a reader to pull MOS devices from subcircuit models rather than from "M" elements. Basically this customization does the opposite part of the writer customization before (only for MOS devices).

```
# Provides a SPICE netlist reader delegate which turns
# some subcircuit models (for subcircuits NMOS and PMOS)
# into devices

class SubcircuitModelsReader < RBA::NetlistSpiceReaderDelegate

  # implements the delegate interface:
  # says we want to catch these subcircuits as devices
  def wants_subcircuit(name)
    name == "NMOS" || name == "PMOS"
  end

  # implements the delegate interface:
  # take and translate the element
  def element(circuit, el, name, model, value, nets, params)
```

```
    if el != "X"
      # all other elements are left to the standard implementation
      return super
    end

    if nets.size != 4
      error("Subcircuit #{model} needs four nodes")
    end

    # provide a device class
    cls = circuit.netlist.device_class_by_name(model)
    if ! cls
      cls = RBA::DeviceClassMOS4Transistor::new
      cls.name = model
      circuit.netlist.add(cls)
    end

    # create a device
    device = circuit.create_device(cls, name)

    # and configure the device
    [ "S", "G", "D", "B" ].each_with_index do |t,index|
      device.connect_terminal(t, nets[index])
    end

    # parameters in the model are given in micrometer units, so
    # we need to translate the parameter values from SI to um values:
    device.set_parameter("W", (params["W"] || 0.0) * 1e6)
    device.set_parameter("L", (params["L"] || 0.0) * 1e6)

    return true

  end

end

# Instantiate a reader using the new delegate
reader = RBA::NetlistSpiceReader::new(SubcircuitModelsReader::new)

# Import the schematic with this reader
schematic("inv_xmodels.cir", reader)
```

Layout-to-Netlist database/report

The layout-to-netlist database (L2N DB) is written using the global report_netlist function. This function can be put anywhere in the script. Writing will happen after the script executed successfully:

```
report_netlist("extracted.l2n")
```

Without the filename, only the netlist browser will be opened but no file will be written. The layout-to-netlist database is a KLayout-specific format. It contains the netlist information plus the shape and instance information from the layout. L2N databases can be read into the netlist browser for example. Hence exchange of extracted netlists is possible.

Layout-vs-Schematic database/report

The Layout-vs-schematic database (LVS DB) is written using the global report_lvs function. This function can be put anywhere in the script. Writing will happen after the script executed successfully:

```
report_lvs("extracted.lvsdb")
```

Without the filename, only the netlist browser will be opened but no file will be written. The LVS database is a KLayout-specific format. It contains the extracted netlist information, the reference netlist and the cross-reference table. LVS databases can be read into the netlist browser for example. Hence exchange of LVS reports is possible.

1.5.7. LVS Connectivity

Intra- and inter-layer connections

The connectivity setup of a LVS script determines how the connections are made. Connections are usually made through conductive materials such as Aluminium or Copper. The polygons representing such a material form a connection. Connections can be made across multiple polygons - touching polygons form connected islands of conductive material. This "intra-layer" connectivity is implicit: in LVS scripts connections are always made between polygons on the same layer.

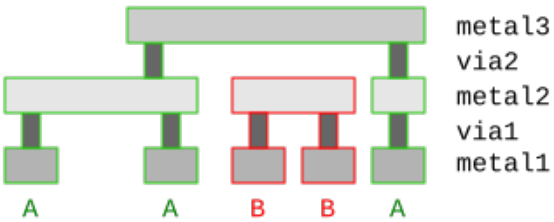
Connections often cross layers. A via for example is a hole in the insulator sheet which connects two metal layers. This connection is modelled using a "connect" statement (see connect):

```
connect(layer1, layer2)
```

A connect statement will specify an electrical connection when the polygons from layer1 and layer2 overlap. layer1 and layer2 are original or derived layers. "connect" statements should appear in the script before the netlist is required - i.e. before "compare" or any other netlist-related statement inside the LVS script. The order of the connect statements is not relevant. Neigther is the order of the arguments in "connect": connections are always bidirectional.

This is an example for a vertical cross section through a simple 3-metal layer stack with the corresponding "connect" statements:

```
connect(metal1, via1)
connect(via1, metal2)
connect(metal2, via2)
connect(via2, metal3)
```



Labels can be included in the connectivity too. Typically labels are placed on metal layers. If the labels are drawn on the same layer than the metal shapes they are automatically included when using "input" to read the layer. If only labels shall be read from a layer, use "labels" (see labels).

To attach labels to metal layers, simply connect the label and metal layers:

```
metal1_labels = labels(10, 0)
metal1        = input(11, 0)
via1          = input(12, 0)
metal2_labels = labels(13, 0)
metal2        = input(14, 0)

connect(metal1, metal1_labels)
connect(metal1, via1)
connect(via1, metal2)
connect(metal2, metal2_labels)
```

If labels are connected to metal layers, their text strings will be used to assign net names to the resulting nets. Ideally, one net is labeled with a single text or with texts with the same text string. In this case, the net name will be non-ambiguous. If multiple labels with different strings are present on a net, the net name will be made from a combination of these names.

Global connections

KLayout supports implicit connections made across all polygons on a layer, regardless whether they connect or not. A typical case for such a connection is the substrate (aka "bulk"). This connection represents the (lightly conductive) substrate material. There is no polygon representing the wafer. Instead, a layer is defined which makes a global connection with "connect_global" (see connect_global):

```
connect_global(bulk, "VSS")
```

The arguments to "connect_global" is the globally connected layer and the name of the global net to create. The function will make all shapes on "bulk" being connected to a single net "VSS". Every circuit will at least have the "VSS" net. In addition, each circuit will be given a pin called "VSS" which propagates this net to parent circuits.

Implicit connections

Implicit connections can be useful to supply preliminary connections which are supposed to be created higher up in the hierarchy: Imagine a circuit with a big power net for example. When the layout is made, the power net may not be completely connected yet, because the plan is to connect all parts of this power net later when the cell is integrated. In this situation, the subcircuit cell itself won't be LVS clean, because the power net is a single net schematic-wise, but exist as multiple nets layout-wise. This prevents bottom-up verification - a very useful technique to achieve LVS clean layouts. It also prevents matching in general, as the layout cell will have two pins while the schematic subcircuit has only one. In this case, the cell and subcircuit will never match.

To allow verification of such a cell, "implicit connections" can be made by giving the net parts the same name through labels and assuming these parts are connected: for example to specify implicit connections between all parts of a "VDD" net, place a label "VDD" on each part and include the following statement in the script:

```
connect_implicit("VDD")
```

"connect_implicit" (see connect_implicit) can be present multiple times to make many of such connections. Implicit connections are accepted on top level, but a warning is issued, indicating that the connection needs to be made further up in the hierarchy. In a subcircuit, implicit connections are required to be connected on the next level of hierarchy - either physically or by another implicit connection. This way, a missing physical connection does not escape and at least a warning is issued if the connection is still not made on top level.

You can declare the layout as being a top level one. This turns the warning about missing physical connections into an error:

```
top_level(true)
```

The "connect_implicit" feature is also called "must connect" nets in other systems.

You can include labels of a certain class in a "connect_implicit" statement using glob-style pattern:

```
connect_implicit("VDD*")
```

This will connect all nets labelled with "VDD1" for example or those labelled with "VDD_5V". However, this statement will only connect "VDD1" with "VDD1", **not** nets with different labels. I.e. it will not connect "VDD1" with "VDD2" labels. To make connections between differently named nets, use "explicit connections" (see below).

"connect_implicit" can be present multiple times. Each statement extends the choice of labels which will be connected.

The above examples of "connect_implicit" apply to all cells. The statement can be made cell specific, by giving a cell name glob pattern for the first argument, followed by the net name pattern.

The following statement will connect all nets labelled with "VDD" from the "MEMCELL" subcell:

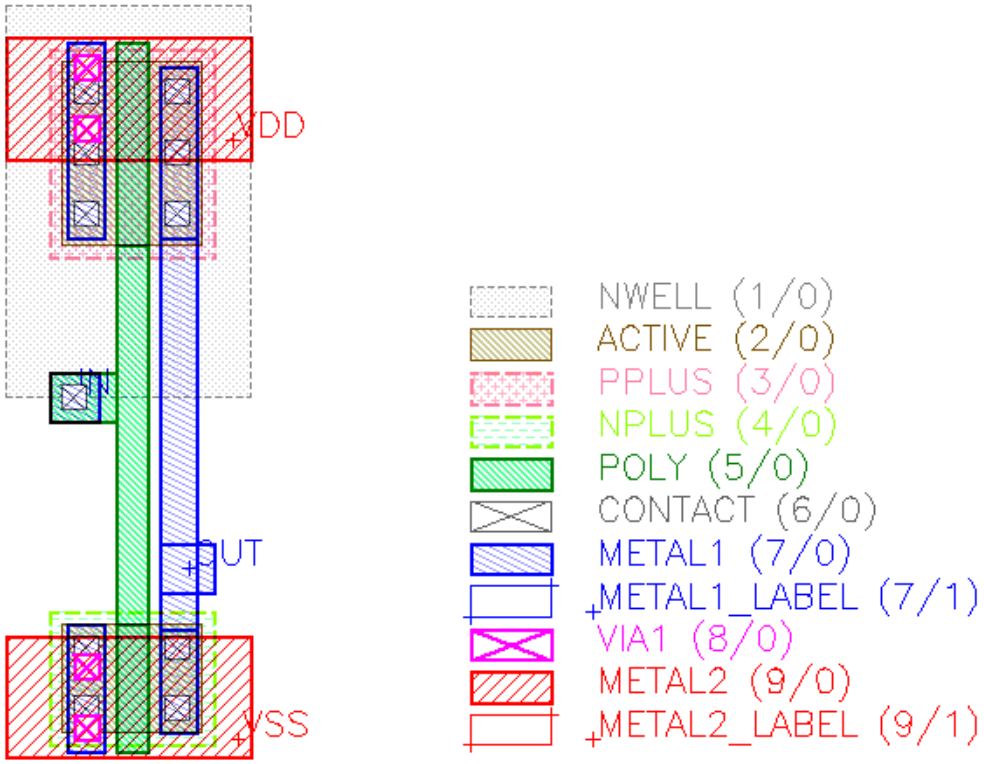
```
connect_implicit("MEMCELL", "VDD")
```

The rule is applied to all cells matching the glob pattern in the first argument. Again, the "connect_implicit" rule may be given multiple times. In this case, all matching occurrences act together.

The "connect_implicit" statements must be given before the netlist is extracted. Typically this happens before or shortly after "connect" statements.

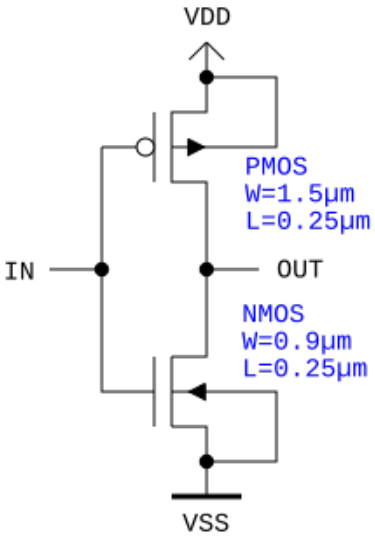
Explicit connections

Explicit connections can be useful to enforce a connection in the layout which is made in the schematic, but not physically on the level of the cell. For example consider the following layout for an inverter:

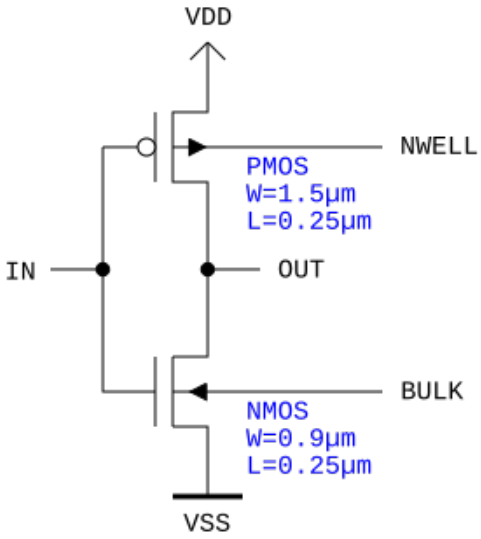


In the layout there are no tie-down diodes, hence there is no physical connection to the n-well region and no physical connection to the bulk substrate. This saves space, but these diodes need to be added by other ways. Usually this is done when the standard cells are combined into macros. Filler cells will be added which include these substrate and well contacts.

On the inverter level however, there is no such connection. Therefore the inverter has separate bulk and n-well pins. The schematic sometimes is a simplified version which does not offer these pins. Hence there is an intrinsic mismatch between layout and schematic.



Inverter schematic



Inverter layout

To align layout and schematic, bulk and VSS pins can be connected explicitly. Same for n-well and VDD. This scheme is similar to the "connect_implicit" scheme explained above, but can connect differently named nets.

To establish an explicit connection in the above example, make sure that n-well and bulk have proper names. For the n-well this can be done by creating labels on the n-well islands giving them a proper name - e.g. "NWELL". The bulk isn't a real layout layer with polygons on it. Using "connect_global" will both connect everything on this layer and give it a name.

The following code will connect the bulk net with "VSS" inside the cell "INV":

```
connect_global(bulk, "BULK")
...
connect_explicit("INV", [ "BULK", "VSS" ])
```

Note that this rule will form a new net called "BULK,VSS" combining both subnets.

The cell name can be a pattern. For example "INV*" will apply this rule on all cells starting with "INV". The cell pattern is not mandatory: if it is omitted, the rule is applied to all cells.

Like implicit connections, explicit connections are checked for being made on the next level of hierarchy, either physically or by another explicit or implicit connection.

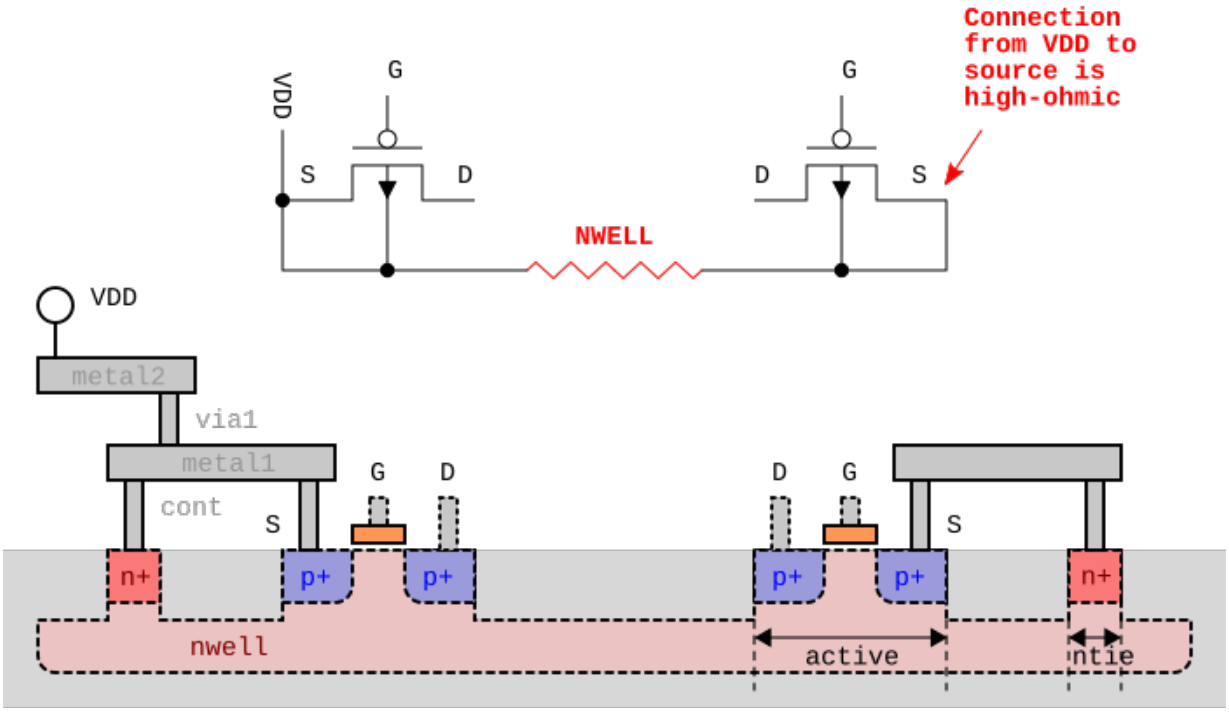
An explicit connection will also imply implicit connections on the nets listed in the net names. So in the example above, different pieces of "VSS" are connected even if they are not physically connected. Again, it is checked that these connections are made later up in the hierarchy.

Again, the "connect_explicit" statements must be given before the netlist is extracted. Typically this happens before or shortly after "connect" statements.

Soft connections

Soft connections are a way to find wiring issues where signals or even power is routed over high-ohmic paths. High-ohmic paths can be established through connections via poly silicon, implant, well or substrate areas. Such areas can easily show resistance values which are a hundred times higher than that of metal connections. We have to make sure that for routing power or critical signals, connections are not made through such areas, but primarily through metal connections.

Here is an example:



In this case, we have a standard textbook planar CMOS technology with two PMOS devices sitting in a n-well. These could be the two PMOS of two inverter pairs for example. Both PMOS need to be connected to VDD at their sources. In addition, the n-well area also needs to be tied to VDD in order to provide reverse bias for the p+ drain areas and the body potential for the transistors, forming the opposite electrode of the gate capacity.

Such a technology stack can be described by the following connectivity:

```
# Input layers

nwell = ...
active = ...
pplus = ...
nplus = ...
poly = ...
contact = ...
metal1 = ...
via1 = ...
metal2 = ...

# computed layers
```

```
(nactive, pactive) = active.and_not(nwell)

# PMOS and NMOS source/drain regions
psd  = (nactive & pplus) - poly
nsd  = (pactive & nplus) - poly

# n tie and p tie (nwell and substrate contact)
ntie = nactive & nplus
ptie = pactive & pplus

# connections

# nwell connections
connect(ntie, nwell)
connect(contact, ntie)

# substrate connections
connect_global(ptie, "BULK")
connect(contact, ptie)

# device connections
connect(contact, psd)
connect(contact, nsd)
connect(contact, poly)

# metal connections
connect(metal1, contact)
connect(via1, metal1)
connect(metal2, via1)
```

However, there is an issue: As shown in the picture, the left PMOS source is properly connected to VDD. The right one however lacks the metal connection to VDD. From the perspective of pure connectivity, this transistor's source is connected to VDD, but only through a weak n-well connection. Such a device will not work - or at least, badly.

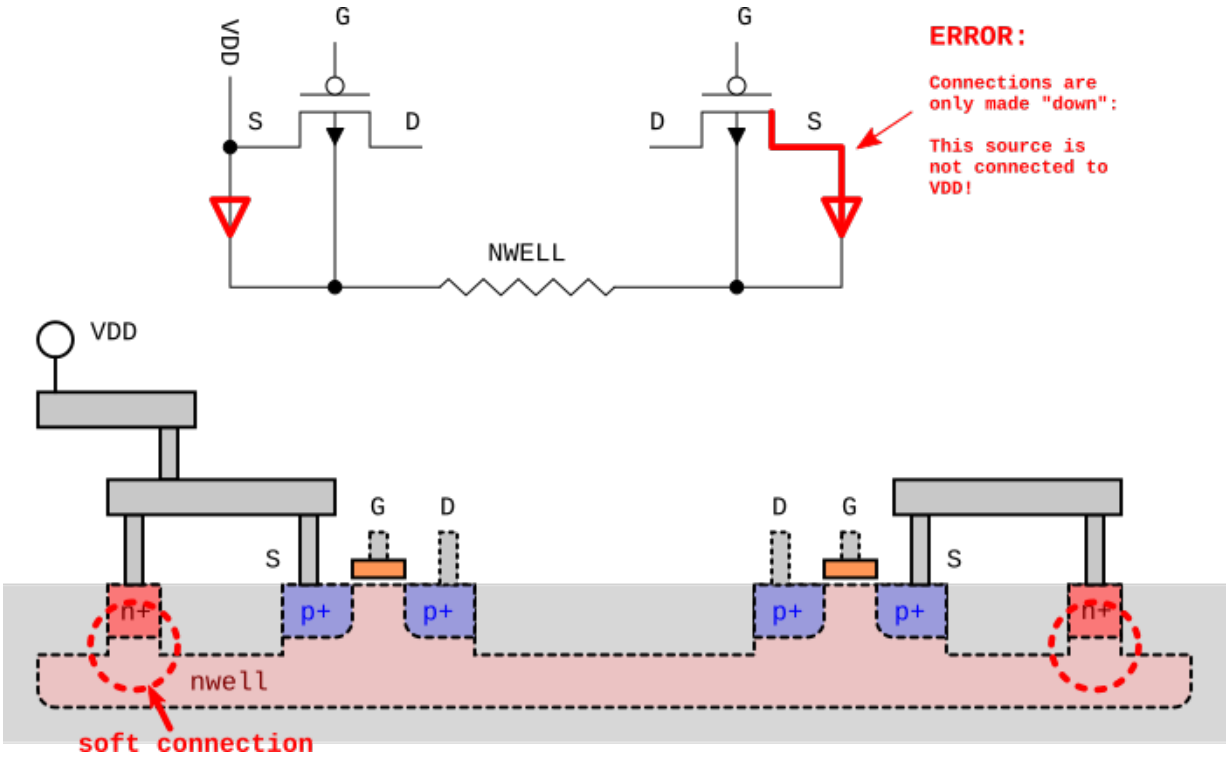
The solution is to introduce soft connections. Soft connections are made by replacing "connect" with "soft_connect" and "connect_global" with "soft_connect_global" (see soft_connect and soft_connect_global). The first layer is the "upper" layer while the second layer is the "lower" layer. The lower layer is the high-ohmic one. In global connections, the global net is always the high-ohmic one.

Soft connections can be visualized as directional connections: current can only flow from the upper to the lower layer, but not the other way. So, a real connection is only made, if both upper terminals of the soft connections are connected to the same physical net.

To solve the n-well issue we have to substitute the n-tie to n-well connection statement by a "soft_connect" statement:

```
soft_connect(ntie, nwell)
```

The above picture now looks like this:



With this definition, the netlist extractor is able to detect the fault and raise a warning or an error (in top level mode). The warning is shown on the log tab and indicates incomplete wiring plus details about the subnets, separated by the soft connections.

The complete the setup we also need to include other soft connections, such as connections via substrate (a global soft connect to the BULK net), via source/drain implants, via the tie implants and via poly:

```
# Input layers

nwell = ...
active = ...
pplus = ...
nplus = ...
poly = ...
contact = ...
metal1 = ...
via1 = ...
metal2 = ...

# computed layers

(nactive, pactive) = active.and_not(nwell)

# PMOS and NMOS source/drain regions
psd  = (nactive & pplus) - poly
nsd  = (pactive & nplus) - poly

# n tie and p tie (nwell and substrate contact)
ntie = nactive & nplus
ptie = pactive & pplus

# connections

# nwell connections
soft_connect(ntie, nwell)
soft_connect(contact, ntie)
```

```
# substrate connections
soft_connect_global(ptie, "BULK")
soft_connect(contact, ptie)

# device connections
soft_connect(contact, psd)
soft_connect(contact, nsd)
soft_connect(contact, poly)

# metal connections
connect(metal1, contact)
connect(via1, metal1)
connect(metal2, via1)
```

As this code demonstrates, multiple soft connections can be specified. From the perspective of the check, all soft connections are of the same kind.

Note, that two opposite soft connections cancel, so this would eventually make a hard connection:

```
soft_connect(a, b)
soft_connect(b, a)
```

NOTE: It is therefore important to observe the direction of soft connections: upper and high-conductive / low-ohmic layer first, and lower and low-conductive / high-ohmic layer second.

Soft connections and "must connect" nets

Soft connections and must connect nets (aka "connect_explicit" and "connect_implicit") serve the same purpose - to detect incomplete wiring. Typically they are used together, such as doing "connect_implicit" on VDD nets and "connect_explicit" on VDD and NWELL nets if the schematic circuits do not feature an explicit NWELL pin.

Soft connections are checked before connect_explicit and connect_implicit are executed. This means, that soft connection errors cannot be masked by declaring them "must connect". On the other hand, that is not a real issue as both checks would raise an warning or error (in the top-level case). It would only be a different kind of warning.

1.5.8. LVS Compare

The actual compare step is rather simple. Provided you have set up the extraction (extract_devices), the connectivity (connect, connect_global, connect_implicit) and provided a reference netlist (schematic), this function will perform the actual compare:

```
compare
```

This method (compare will extract the netlist (if not already done) and compare it against the schematic. It returns true on success and false otherwise, in case you like to take specific actions on success or failure.

The compare step can configured by providing hints.

Net equivalence hint

It can be useful to declare two nets as identical, at least for debugging. The compare algorithm will then be able to deduce the real causes for mismatches. It is helpful for example to provide equivalence for the power nets, because netlist compare fails will often cause the power nets no to be mapped. This in turn prevents matching of other, good parts of the circuit. To supply a power net equivalence for "VDD" within a circuit (e.g. "LOGIC"), use this statement:

```
same_nets("LOGIC", "VDD", "VDD:P")
```

In this example it is assumed that the power net is labeled "VDD" in the layout and called "VDD:P" in the schematic. Don't leave this statement in the script for final verification as it may mask real errors.

"same_nets" can also be used to require a matching between specific nets. This is useful on top level to check for matching nets assigned to specific pads. This allows checking correct pad assignment. For example to check whether the same net is attached to the "VDD" pad, label the net "VDD" in the layout and specify:

```
same_nets!("CHIP", "VDD", "VDD")
```

The exclamation-mark version will report a net mismatch if either there is no "VDD" net in either layout or schematic or if these nets to not match. The above specification can be abbreviated as layout and schematic net name are identical:

```
same_nets!("CHIP", "VDD")
```

It's also possible to specify pattern for circuit names or net names. This example requires all nets starting with "PAD" to have a counterpart in layout and schematic for circuit "TOP" and each of these pairs has to match:

```
same_nets!("TOP", "PAD*")
```

So it is an error if there is a PAD1 net in layout but none in the schematic. It is also an error if a net called PAD2 is there is layout and schematic but they do not match.

"same_nets" and "same_nets!" can appear anywhere in the LVS script.

For more information about "same_nets" see same_nets and same_nets!.

Circuit equivalence hint

By default, circuits with the same name are considered equivalent. If this is not the case, equivalence can be established using the same_circuit function:

```
same_circuits("CIRCUIT_IN_LAYOUT", "CIRCUIT_IN_SCHEMATIC")
```

Declaring circuits as 'same' means they will still be compared. The function is just a hint where to look for the compare target.

Device class equivalence hint

By default, device classes with the same name are considered equivalent. If this is not the case, equivalence can be established using the same_device_classes function:

```
same_device_classes("PMOS_IN_LAYOUT", "PMOS_IN_SCHEMATIC")
same_device_classes("NMOS_IN_LAYOUT", "NMOS_IN_SCHEMATIC")
```

This method can be used also multiple times to establish a many-to-many equivalence:

```
same_device_classes("POLYRES", "RES")
same_device_classes("WELLRES", "RES")
```

If one target is "nil", the corresponding devices are basically ignored:

```
# ignores "POLYRES" devices:
same_device_classes("POLYRES", nil)
```

Tolerances

When comparing device parameters, by default strict equivalence is required. However, when drawing a device like a resistor, it's usually difficult to match the exact value unless the resistor calibration is consistent with drawing grids and the resistor geometry is not confined by design rule constraints. So sometimes the target value or a device parameter can only be approximated in the layout. This will by default lead to a mismatch.

The solution is to specify parameter tolerances. Tolerances can be specified in an absolute or relative fashion. If an absolute tolerance is given, the layout parameter may deviate from the target value by this tolerance either to lower or higher values. So the unit of the tolerance is the same than the unit of the parameter.

If a relative tolerance is given, the deviation is computed from the target value times the tolerance. So the relative tolerance is a factor and a value of 0.05 for example specifies an allowed deviation of plus or minus 5%. Relative tolerances are unit-less.

It's also possible to specify both an absolute and a relative tolerance. In this case, both tolerances add and the allowed deviation becomes larger.

To specify an absolute tolerance, use the tolerance function:

```
tolerance("NMOS", "L", 0.05)
```

The two arguments are the name of the device class and the name of the parameter for which the tolerance will be applied. In the case above, a tolerance of 50nm (the unit of L is micrometer) is applied to the length parameter of "NMOS" devices.

A relative tolerance is specified as an additional forth parameter. You can set the absolute tolerance to zero to specify only relative tolerances. This will specify 1% tolerance for the "L" parameter of "NMOS" devices:

```
tolerance("NMOS", "L", 0.0, 0.01)
```

There is also a more explicit notation for the tolerance:

```
tolerance("NMOS", "L", :absolute => 0.05)
```

or

```
tolerance("NMOS", "L", :relative => 0.01)
```

An absolute plus relative tolerance can be specified by giving both. The following calls will give you 50nm absolute and 1% relative tolerance for the "L" parameter of "NMOS" devices:

```
tolerance("NMOS", "L", 0.05, 0.01)
tolerance("NMOS", "L", :absolute => 0.05, :relative => 0.01)
```

Ignoring parameters

It is possible to ignore certain parameters from certain devices in the netlist compare. For example, if you don't want to compare the "L" parameter of the "NMOS" devices, use this statement:

```
ignore_parameter("NMOS", "L")
```

This statement can be put into the script anywhere before the "compare" statement.

By default, only "primary" parameters are compared. For a resistor for example, "R" is a primary parameter, the other ones like "L", "W", "A" and "P" are not. Using "tolerance" will implicitly enable a parameter - even if it is not a primary one - while "ignore_parameter" will disable a parameter for compare - even if it is a primary one.

Enabling and disabling parameters

As mentioned before, some device parameters are primary while other are not. For example, for the resistor device, "R" (the resistance value) is a primary parameter while the device length ("L") is not. You can make the "L" parameter primary for a device class called "RES" by using:

```
enable_parameter("RES", "L")
```

This has two effects: first, the "L" parameter is written into the Spice output netlist and in addition it is compared against the schematic "L" parameter.

Correspondingly, a primary parameter can be disabled using:

```
disable_parameter("RES", "R")
```

This behavior is overridden by a "tolerance" or "ignore_parameter" specification for that parameter or if a custom device comparer is installed. Netlisting is affected only for the elementary devices (R, C and L) and any Spice writer delegate can choose to ignore the primary flag. A custom device comparer may also ignore this flag. So after all, enabling or disabling a parameter is not a strong concept but rather a hint.

Pin swapping

Pin swapping can be useful in cases, where a logic element has logically equivalent, but physically different inputs. This is the case for example for a CMOS NAND gate where the logic inputs are equivalent in function, but not in the circuit and physical implementation. For such circuits, the compare function needs to be given a degree of freedom and be allowed to swap the inputs. This is achieved with the equivalent_pins function:

```
equivalent_pins("NAND_GATE", "A", "B")
```

The first argument is the name of the circuit in the layout netlist. You can only specify equivalence in layout, not in the reference schematic. Multiple pins can be listed after the circuit name. All of them will be considered equivalent.

Capacitor and resistor elimination

This feature allows eliminating "open" resistors and capacitors. Serial resistors cannot be elimiated currently (shorted).

To eliminate all resistors with a resistance value above a certain threshold, use the max_res function. This will eliminate all resistors with a value >= 1kOhm:

```
max_res(1000)
```

To eliminate all capacitors with a capacitance value below a certain threshold, use the min_caps function. This will eliminate all capacitances with a value <= 0.1fF:

```
min_caps(1e-16)
```

Compare and netlist hierarchy

Good layouts are built hierarchically and the netlist compare can make use of hierarchy. "Hierarchically" means that a circuit is built from cells which itself map to subcircuits of the schematic netlist. The netlist extractor tries hard to maintain the hierarchy and the netlist compare will utilize the hierarchy to provide more meaningful reports and enable a bottom-up design approach.

Given a hierarchical layout and schematic netlist, the compare algorithm will work bottom-up: it will first compare the leaf circuits (circuits without subcircuit calls) and if those match, it will continue with the calling circuits. This approach is more efficient and fosters a clean relationship between layout and schematic netlist.

To enable hierarchical extraction, you must use "deep" mode (deep). If the deep mode statement is missing, the layout netlist will be flat (i.e. without subcircuits).

The second useful feature is "align" (align). This statement will remove circuits from the layout or schematic netlist which are unknown in the other netlist. Often, layouts contain helper cells which are not corresponding to a circuit (e.g. via cells). These are removed in this step. Eventually, this step will also flatten the schematic netlist if the layout has been extracted in a flat way.

In general, it's a good idea to include "align" before "netlist.simplify" or similar netlist manipulation and the "compare" step.

A very useful side effect of "align" is this: it will remove circuits above the top level circuit of either side. So it will eventually render a sub-tree from the circuit tree and use that for compare. This enables **subcell verification**: by selecting a subcell in the layout hierarchy, an "align"-enabled LVS script will compare this cell against the corresponding subcircuit in the schematic netlist. It will ignore the parent hierarchy of this subcircuit. This way, you can work yourself upwards in the hierarchy and fix LVS errors cell by cell with the same schematic netlist.

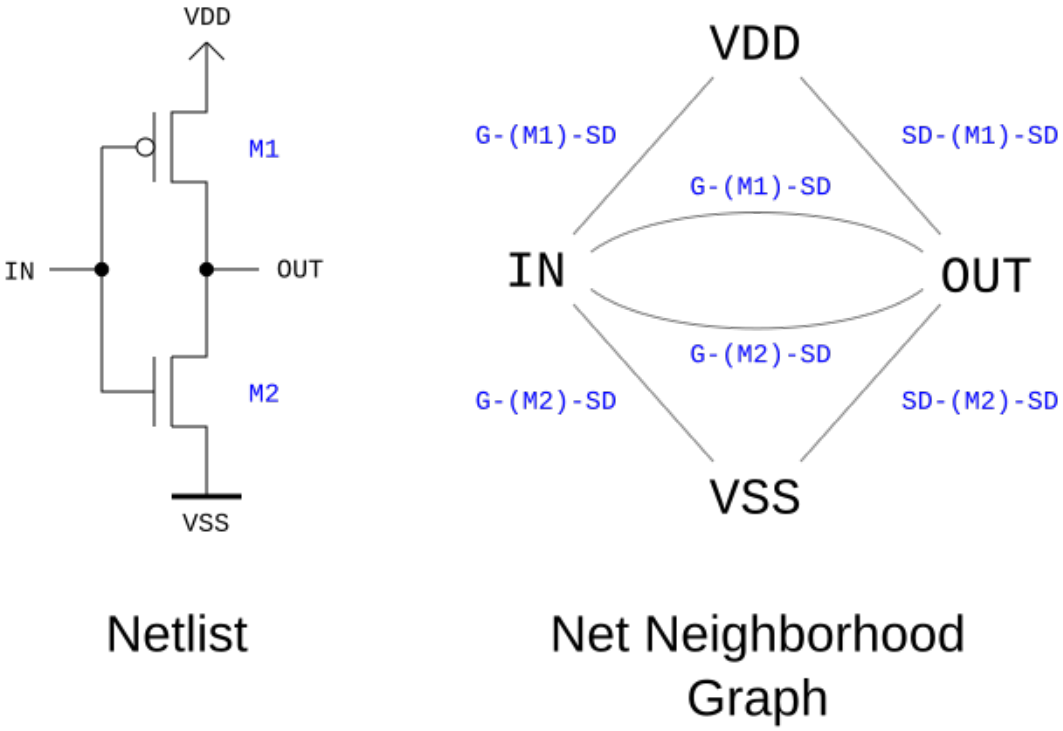
How the compare algorithm works

The coarse flow of the netlist compare algorithm is this:

```
foreach circuit bottom up:
  if matching circuit found in reference netlist:
    if all subcircuits have been matched and pin matching has been established for them:
      compare net graph locally from this circuit
    else:
      skip circuit with warning
  else:
    issue a circuit mismatch error
```

A consequence of this flow is that the compare will stop treating parent circuits when one circuit's pins can't be matched to pins from the corresponding reference circuit or the corresponding circuit can't be found in the reference netlist. This behaviour fosters a bottom-up debugging approach: first fix the issues in subcircuits, then proceed to the parent circuits.

The local net graph compare algorithm is a backtracking algorithm with hinting through topological net classification. Topological net classification is based on nearest-net neighborhood. The following image illustrates this:



Here the IN net's neighborhood is VDD via a traversal of gate to source/drain over M1, to OUT via a twofold traversal of gate to source/drain over M1 and M2 and to VSS via another single traversal of gate to source/drain over M2. This uniquely identifies IN in this simple circuit. In effect, OUT, VDD and VSS can be identified uniquely because their transitions from the IN net are unambigously identifying them. The topological neighborhood is a simple metrics which allows identifying matching nets from two netlists and deducing further relations.

In big netlists, the algorithm will first try to match nets unambigously according to their neighborhood metrics and register them as paired nets. Such pairs often allow deducing further matching pairs. This deduction is continued until all non-ambiguous pairing options are exhausted. For resolving ambiguities, backtracking is employed: the algorithm proposes a match and tentatively proceeds with this assumption. If this execution path leads to a mismatch or logical contradiction, the algorith will go back to the beginning and restart with a new proposal. Backtracking is usually required mainly to match networks with a high symmetry such as clock trees.

1.5.9. LVS Netlist Tweaks

Netlist tweaking is important to standardize netlists. Without tweaking, the extracted netlist may contain elements that are redundant or don't match anything found in the schematic.

Netlist tweaks are applied on the extracted Netlist object. This can be obtained with the netlist function. This function will extract the netlist if not done already.

Netlist tweaks can also be applied to the schematic netlist. For example to flatten away a model subcircuit called "NMOS", use this Netlist#flatten_circuit:

```
schematic.flatten_circuit("NMOS")
```

Top level pin generation

Circuits extracted don't have pins on the top hierarchy level as the extractor cannot figure out where to connect to this circuit. The compare function does not try to match pins in this case. But to gain a useful extracted netlists, pins are required. Without pins, a circuit can't be embedded in a testbench for example.

KLayout offers a function to create top-level pins using a simple heuristics: for every named (i.e. labeled) net in the top level circuit a pin will be created (Netlist#make_top_level_pins):

```
netlist.make_top_level_pins
```

Device combination

Combining devices is important for devices which are not represented as coherent entities in the layout. Examples are:

- **Fingered MOS transistors:** MOS transistors with a large width are often split into multiple pieces to reduce the parasitic gate and diffusion resistances and capacitances. In the layout this is equivalent to multiple parallel transistors.
- **Serial resistors:** Large resistors are often separated into stripes which are then connected in a meander structure. From the device perspective such resistors consist of several resistors connected in series.
- **Array capacitors:** Large capacitors are often split into smaller ones which are arranged in an array and connected in parallel. This helps controlling the parasitic series resistances and inductances and avoids manufacturing issues.

In all these cases, the schematic usually summarizes these devices into a single one with lumped parameter values (total resistance, capacitance, transistor width). This creates a discrepancy which "device combination" avoids. "Device combination" is a step in which devices are identified which can be combined into single devices (such as serial or parallel resistors and capacitors). To run device combination, use Netlist#combine_devices:

```
netlist.combine_devices
```

The combination of serial devices might leave floating nets (the net connecting the devices originally. These nets can be removed with Netlist#purge_nets. See also Netlist#simplify, which is wrapper for several methods related to netlist normalization.

It's recommended to run "make_toplevel_pins" and "purge" before this step (see below).

Circuit flattening (elimination)

It's often required to flatten circuits that do not represent a specific level of organisation but act as a wrapper to something else. In layouts, devices are often implemented as PCells and appear as specific cells for no other reason than being implemented in a subcell. The same might happen for schematic subcircuits which wrap a device. "Flattening" means that a circuit is removed and its contents are integrated into the calling circuits.

To flatten a circuit from the extracted netlist use Netlist#flatten_circuit:

```
netlist.flatten_circuit("CIRCUIT_NAME")
```

The argument to "flatten_circuit" is a glob pattern (shell-like). For example, "NMOS*" will flatten all circuits starting with "NMOS".

Automatic circuit flattening (netlist alignment)

Instead of flattening circuits explicitly, automatic flattening is provided through the align method.

The "align" step is optional, hence useful: it will identify cells in the layout without a corresponding schematic circuit and flatten them. "Flatten" means their content is replicated inside their parent circuits and finally the cell's corresponding circuit is removed. This is useful when the layout contains structural cells: such cells are inserted not because the schematic requires them as circuit building blocks, but because layout is easier to create with these cells. Such cells can be PCells for devices or replication cells which avoid duplicate layout work.

The "align" method will also flatten schematic circuits for which there is no layout cell:

```
align
```

Black boxing (circuit abstraction)

Circuit abstraction is a technique to reduce the verification overhead. At an early stage it might be useful to replace a cell by a simplified version or by a raw pin frame. The circuits extracted from such cells is basically empty or are intentionally simplified. But as long as there is something inside the cell which the parent circuit connects to, pins will be generated. These pins then can be thought of as the circuit's abstraction.

A useful method in this context is the "blank_circuit" method. It clears a circuit's innards from a netlist. After this, the compare algorithm will identify both circuits as identical, provided they feature the same number of pins. Named pins are required to match exactly unless declared equivalent. Unnamed pins are treated as equivalent. To name pins use labels on the pin's nets inside the circuit's layout.

To wipe out the innards of a circuit, use the Netlist#blank_circuit method:

```
netlist.blank_circuit("CIRCUIT_NAME")
schematic.blank_circuit("CIRCUIT_NAME")
```

NOTE: In this version, use "blank_circuit" before "purge" or "simplify" (see below). "blank_circuit" sets a flag (Circuit#dont_purge) which prevents purging of abstract circuits.

There is a short form for this too (blank_circuit). In contrast to netlist-based "blank_circuit", this method can be used anywhere in the LVS script:

```
blank_circuit("CIRCUIT_NAME")
```

The argument to "blank_circuit" in both cases is a glob pattern (shell-like). For example, "MEMORY*" will blank out all circuits starting with the word "MEMORY".

Joining of symmetric nodes

Sometimes it is possible to omit connections in the layout because these will not carry any current. This might simplify the layout and allow denser packing, but finally there is a mismatch between schematic and layout. In general, connections can be omitted if they would connect symmetric nodes. When symmetric nodes are swapped, the circuit will not change. Hence they will always carry the same potential (at least in theory) and a connection between them will not carry any current. So it can be omitted.

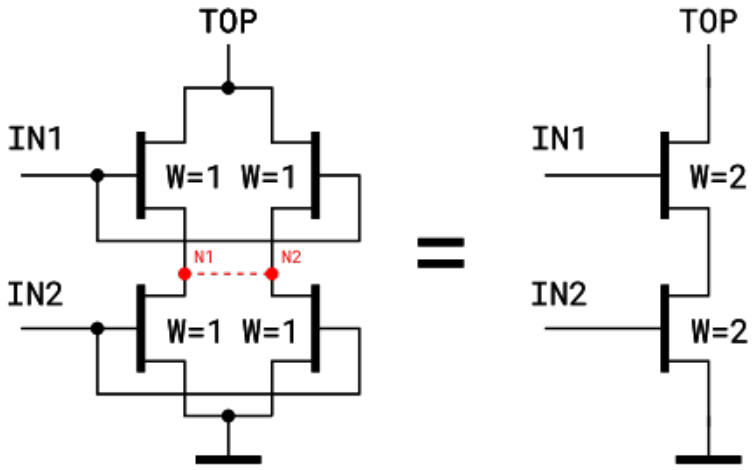
This feature can be used to solve the "split_gates" problem (see "split_gates" below). The internal source/drain nodes are symmetric in the configuration shown there, so "join_symmetric_nets" can be used to solve make the required connections, e.g.:

```
join_symmetric_nets("NAND2")
```

However, there is a more specific feature available ("split_gates") which covers more cases, but is specialized on MOS devices.

Split gates

The following picture describes such a situation known as "split gate configuration". In this case, the N1 and N2 are identical: swapping them will not change the circuit's topology. Hence, they will carry the same potential and the red connection is not required physically. But without such a connection, the parallel transistors (top pair and bottom pair) will not be recognized as parallel and the pairs will not be joined into one each:



KLayout provides a feature (split_gates) which will add such connections after extraction of the netlist:

```
split_gates("NMOS")
```

This function will analyze all circuits in the extracted netlist with respect to "NMOS" devices and connect all split gates relevant source/drain nodes inside. If this function is called before "combine_devices" (e.g. through "netlist.simplify"), this connection is already present then and parallel devices will be recognized and combined.

The device name must denote a MOS3, MOS4, DMOS3 or DMOS4 device. The gate lengths of all involved devices must be identical. For MOS4 and DMOS4, all devices on one gate net must share the same bulk net.

In addition to the device name, a glob-style circuit pattern can be supplied. In this case, the analysis is restricted to the circuits matching this pattern.

"split_gates" can be used anywhere in the LVS script.

Purging (elimination of redundancy)

Extracted netlists often contain elements without a functional aspect: via cells for example generate subcircuits with a single pin and no device. Isolated metal islands (letters, logos, fill/planarisation patches) will create floating nets etc. Two methods are available to purge those elements.

Netlist#purge will remove all floating nets, all circuits without devices or subcircuits. Netlist#purge_nets will only purge floating nets. Floating nets are nets which don't connect to any device or subcircuit.

```
netlist.purge
netlist.purge_nets
```

Normalization wrapper (simplification)

Netlist#simplify is a wrapper for "make_top_level_pins", "purge", "combine_devices" and "purge_nets" in this recommended order:

```
netlist.simplify
```

As a technical detail, "make_top_level_pins" is included in this sequence as with pins, nets are not considered floating. So "purge_nets" will maintain pins for labeled nets even if these nets are not connected to devices. This allows adding optional pins while maintaining the top level circuit's interface.

2. Various Topics

This category covers various special topics and provides additional information for certain features in KLayout. The following topics are available:

2.1. Layer Mapping Tables

Layer mapping tables are used to specify the behavior of the layout reader. Specifically, they define what actions are taken when a shape on a certain layer is encountered. They can be used to

- Filter layers
- Supply layer names for GDS layer and datatype combinations
- Map layers to different ones
- Combine different layers into the same layer

A layer mapping table consists of two specifications: a match expression and an optional target expression. If a shape is encountered, the reader looks up the shape's layer in the mapping table. If an entry is found whose match expression matches the layer of the shape, the shape is assigned a new layer given by the target expression if present. By assigning different match expressions the same target expressions, multiple input layers can be combined into a single one.

If no matching entry is found, the reader can be configured to either store the shape under the original layer or discard it. This option can be found in the reader options dialog as the "Read all layers" option. If that box is checked, the shapes are stored under their original layer and discarded if not.

A target expression can be used also to add information, specifically a layer name. In GDS there is no layer name but just a layer and datatype number. In OASIS, there is a layer name in addition. Other formats just use named layers and don't have the concept of layer number or datatype number. When the target expression specifies a layer name that name is used. That allows adding of OASIS layer names to GDS files for example. Layers with names are usually more useful than layers that just have a number.

The layer mapping table consists of lines, each specifying the match expression and optional target expression. The match and target expressions are separated by a colon. Each expression has the form "layer" (numeric), "layer/datatype" (both numberic), "name" (a string) or "name(layer)" or "name(layer/datatype)" (all specifications). When name and layer/datatype are specified in a match string, KLayout will first look for a matching layer/datatype and then for a matching layer name. The name match is case sensitive. For the numerical specifications, ranges are allowed using a hyphen for an interval and the comma for enumerations (see second example below).

Here are some examples:

1/0 or **1**	Matching layer 1, datatype 0
17/1-5,10	Matching layer 17, datatypes 1 to 5 and 10
1/0:22	Matching Layer 1, datatype 0. Shapes are shifted to layer 22, datatype 0
1/0:A	Matching layer 1, datatype 0. The name "A" is added to that layer
1/0:A(2/0)	Matching Layer 1, datatype 0 mapped. Shapes are shifted to layer 2, datatype 0 and name "A"
A	Matching named layer "A"
A:1/0	Matching named layer "A". Shapes are shifted to layer 1, datatype 0

When the layer mapping is read from a file, each line corresponds to one entry. Blanks are ignored as are empty lines. Comments can be inserted using the "#" character in front of the comment.

Wildcards

Source layers can be specified using wildcards. A wildcard is a "*" character matching "any layer". Examples for such expressions are:

10-*/0	Matching layer 10 and above, datatype 0
***/10**	Matching datatype 10 of every layer
0-5,10-*/*	Matching layer 0 to 5 (inclusive) and above 10, all datatypes.

When ranges or wildcards are used as match expressions, the specified layers will be lumped together into a single layer. This layer will have the least permitted layer and datatype number. For example, with a match expression of "1-10/*", all these layers will be mapped to "1/0". This behavior can be modified using a target layer specification with wildcards.

Wildcard expansion and relative layer mapping

If the match expression includes a numerical range or wildcards for the layer or datatype number, by default all these layers will be combined into a single one, where its layer or datatype number is derived from the least permitted number.

This behavior can be modified using wildcard expansion. This is a target layer which includes a "*" wildcard. This wildcard is substituted by the actual layer or datatype number:

10-*/0 : */10	Maintain layers for layer 10 and above and map datatype to 10
10-*/0 : */*	Select layers 10 and above, datatype 0 and maintain these as individual layers
1/* : 2/*	Map layer number 1 to 2, maintain all datatypes

Relative layer mapping allows adding an offset to the layer or datatype numbers. This offset can be negative with undefined behavior when the resulting number goes below zero:

10-20/*: *+1000/*	Selects all layers between 10 and 20, all datatypes. These layers will be read into the original layers plus 1000 for the layer number.
10/10-*: */*-10	Selects layer 10, datatypes 10 plus. The resulting datatypes will be 10 less starting from 0.

Multi-mapping and unmapping

Layer mapping table support an advanced feature which is to duplicate input layers to a number of output layers (1:n) mapping. The feature is enabled by prepending a "+" to the mapping statement. The following statement will first select layer 5/0 and additionally copy it to layer 1000/0:

```
5/0
+5/0: 1000/0
```

Unmapping removes the mapping for a specific layer or range. It is specified by prepending "-" to the mapping expression. The following statement will map all datatypes of layer 5 to 0 except for datatype 10 which is not considered.

```
5/*: 5/0
-5/10
```

Unmapping cancels the mappings specified previously, so the order of statements becomes important when using unmapping and multi-mapping.

Brackets

Square brackets can be used to imply mapping to the original layer. When putting square brackets around a mapping expression, the default target is "*/*", which means expansion to the original layer. Hence the following statements are identical:

```
[1-10/*]
1-10/* : */*
```

When combining this with "+" for multi-mapping, put "+" in front of the bracket.

You can put round brackets around mapping expressions for visual clarity, specifically when combining them with "-" (unmapping) or "+" (multi-mapping):

```
-(1-10/*)
+(17/0 : 1017/0)
```

2.2. About Layer Specifications

Layer specifications are used in various places, for example in layer mapping files (Layer Mapping Tables). Layer specifications are used inside the database to give a layer a name or a number/datatype pair or both. Layer specifications are the text representation of LayerInfo objects.

Blanks within layer specifications are ignored and can be put between the different components of the specification.

A simple number for the specification will indicate a layer with this layer number and a datatype of zero:

```
17
```

will give layer 17, datatype 0.

A number followed by a slash and another number will indicate a layer number and datatype:

```
17/5
```

will give layer 17, datatype 5.

Layers can be named. Named layers are present in DXF, CIF or other formats which don't use the GDS layer/datatype number scheme. Just giving a name will indicate such a layer:

```
METAL1
```

will give a named layer called "METAL1".

If you want to use a name that is a number, use quotes:

```
"17"
```

will give a named layer called "17".

If you want to use a name that includes blanks, put it into quotes as well:

```
"METAL 1"
```

will give a named layer called "METAL 1" (however, such layer names are usually illegal).

Finally, a layer can have both a name and layer/datatype numbers. In this case, add the layer/datatype number to the name in round brackets:

```
METAL1 (17)
```

will give a layer named "METAL1" with layer 17 and datatype 0 and

```
METAL1 (17/5)
```

will give a layer named "METAL1" with layer 17 and datatype 5.

Layer specifications as targets

When used in a target context (e.g. for layer mapping), a layer specification can use wildcards and relative layer/datatype specifications. Using "*" instead of a layer or datatype number means to reuse the source layer or datatype number. Using "+x" or "-x" for layer or datatype number means to add or subtract "x" from the source layer or datatype number.

2.3. Transformations in KLayout

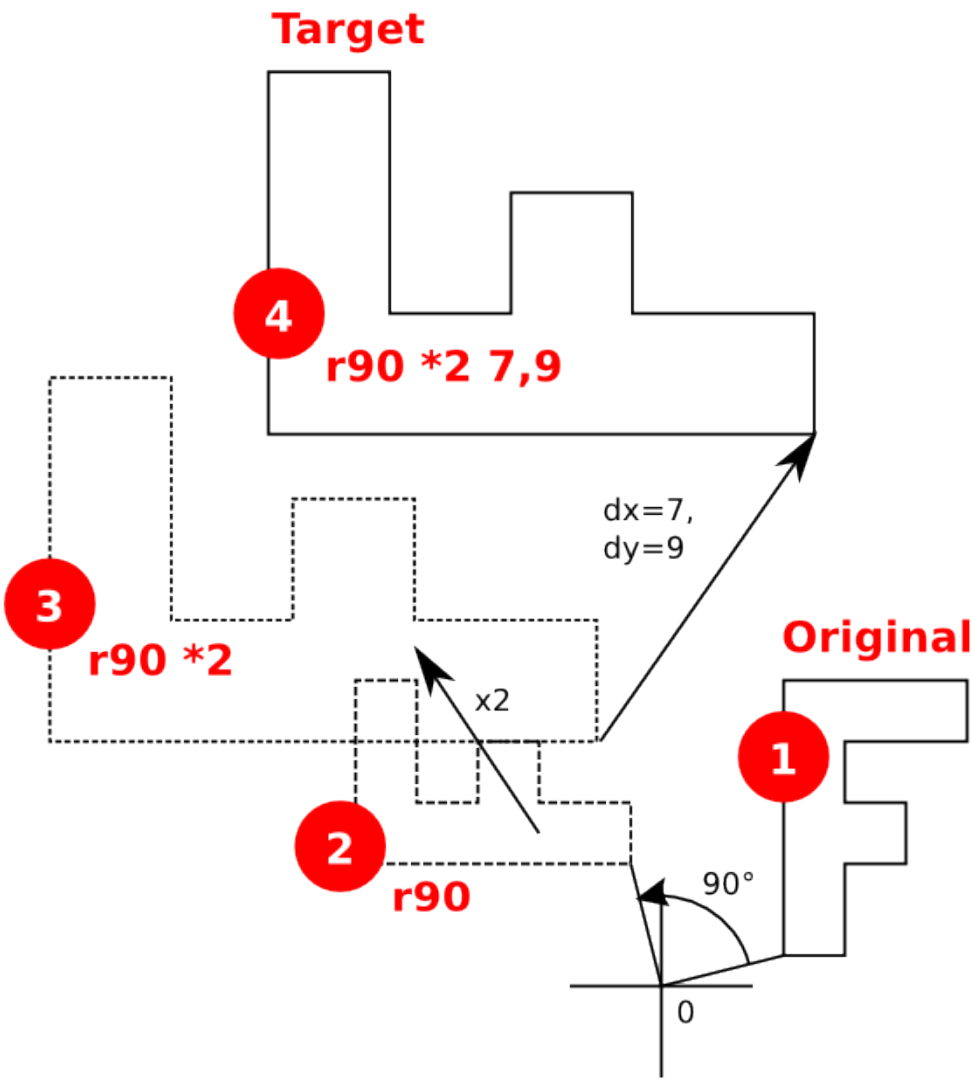
KLayout supports a subset of affine transformations with the following contributions:

- **Rotation and/or mirroring:** rotation by a given angle or mirroring at a given axis.
- **Scaling:** magnification by the given factor.
- **Translation:** a displacement by the given vector.

The execution order is "displacement after rotation, mirroring and scaling". Transformations are used for example to describe the instantiation of a cell. The content of a cell appears in the parent cell after the given transformation has been applied to the content of the cell.

The transformations supported by KLayout cover the transformations employed within GDS2, OASIS and other layout formats. KLayout does not support shearing currently.

The following figure illustrates the effect of the transformation "r90 *2 7,9". This notation specifies a transformation composed of a rotation by 90 degree, a scaling with factor 2 and a displacement by 7 units in x- and 9 units in y-direction. In that example, the "F" shape is first rotated by 90 degree around the origin. Because the "F" is already displaced from the origin, this will also move the "F" shape. The shape then is scaled. Again it will move because every point of the polygon moves away from the origin. Finally it is displaced by the given displacement vector.



The notation shown here is used in many places within KLayout. It is basically composed of the following parts which are combined putting one or more blanks in between. The order the parts are specified is arbitrary: the displacement is always applied after the rotation.

- <x>**,**<y>: A displacement (applied after rotation and scaling) in micron units. If no displacement is specified, "0,0" is assumed.
- **r**<a> or **m**<a>: A rotation by angle "a" (in degrees) or mirroring at the "a" axis (the x axis rotated by "a" degree). If no rotation or mirroring is specified, no rotation is assumed.
- ***<s>: A scaling by the factor "s". If no scaling is specified, no scaling is assumed.

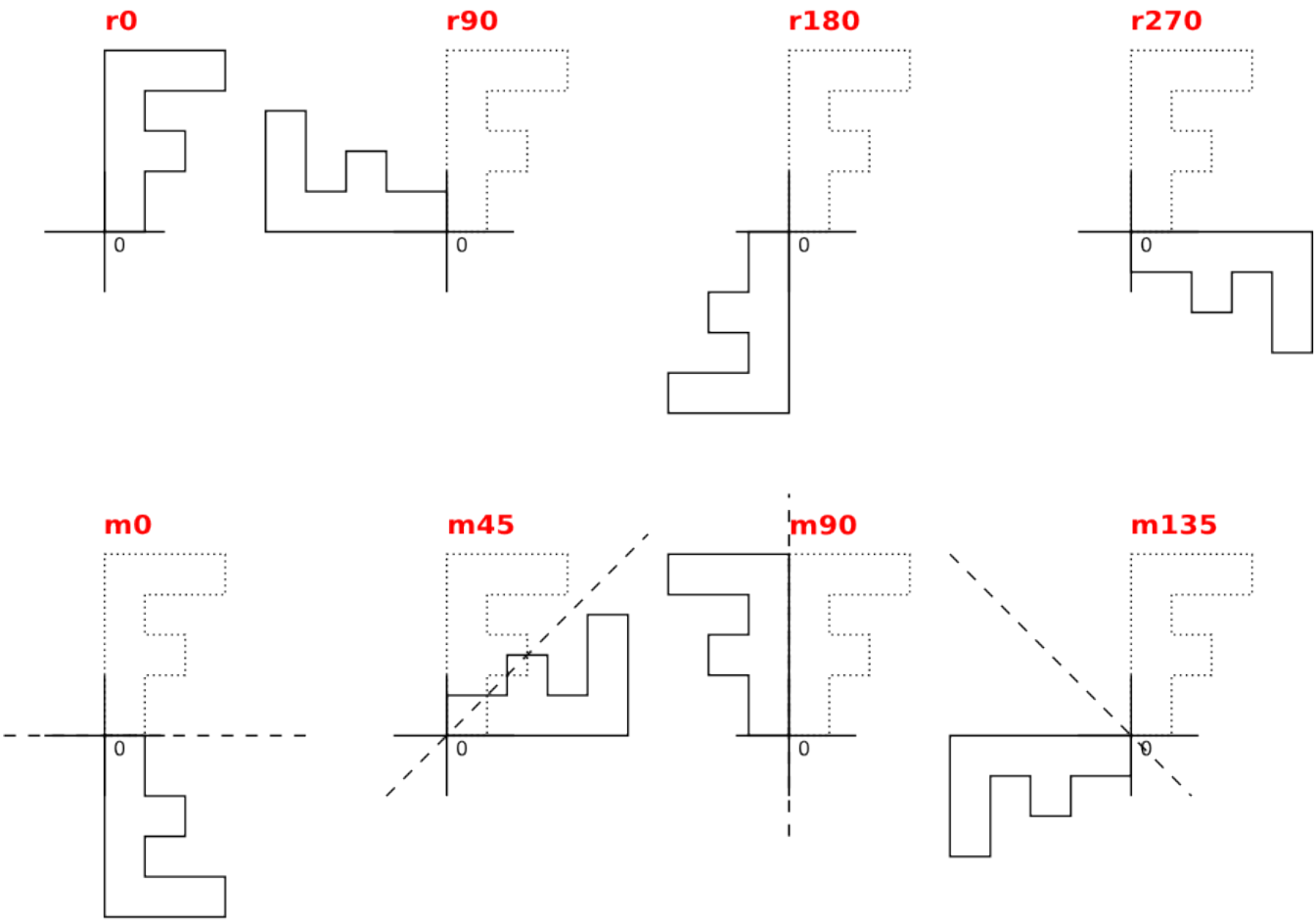
Here are some examples:

- **0,100**: shift 100 units up.
- **r90**: rotation by 90 degree counterclockwise (positive in the mathematical sense).

- **m0**: mirroring at the x-axis.
- **m45 100,-200**: swap x and y (mirror at 45 degree axis), shift 100 units to the right and 200 units down.
- **r22.5 *1.25**: rotate by 22.5 degree and scale by factor 1.25.

The distance units are usually micron. In some cases (i.e. transformations inside a database), the unit is database units and dx and dy are integer values.

Mirroring and rotation are exclusive and mirroring includes a rotation. In fact, a mirror operation at a certain axis is identical to a mirror operation at the x-axis, followed by a rotation by twice the angle "a". The following figure illustrates rotation and mirroring with the eight basic transformations involving rotations by multiples of 90 degree:



KLayout is not restricted to these basic operations. Arbitrary angles are supported (i.e. "r45" or "m22.5"). Usually however, this implies grid snapping and other issues. This also is true for arbitrary scaling values. KLayout is also more effective when using simple transformations involving only rotations by multiples of 90 degree and do not use scaling.

Coding transformations

Note that mirroring at an axis with a given angle "a" is equivalent to mirroring at the x axis followed by a rotation by twice the angle "a". For example:

```
m45 == m0 followed by r90
```

When coding transformations, two parameters are used to represent the rotation/mirror part: a rotation angle and a flag indicating mirroring at the x axis. The mirroring is applied before the rotation. In terms of these parameters, the basic transformations are:

Rotation angle (degree)	Mirror flag = False	Mirror flag = True
0	r0	m0
90	r90	m45
180	r180	m90
270	r270	m135

Transformation objects

Transformation objects are convenient objects to operate with. They represent a transformation (technically a matrix) consisting of an angle/mirror and a displacement part. They support some basic operations:

- Concatenation: `T = T1 * T2`
  `T` is transformation `T2` applied, then `T1` (note this order)
- Inversion: `TI = T.inverted()`
  `TI` is the inverse of `T`, i.e. `TI * T = T * TI = 1` where `1` is the neutral transformation which does not modify coordinates
- Application to geometrical objects: `q = T * p`
  where `p` is a box, polygon, path, text, point, vector etc. and `q` is the transformed object

Vectors and Points

In KLayout there are two two-dimensional coordinate objects: the vector and the point. Basically, the vector is the difference between two points:

```
v = p2 - p1
```

Here `v` is a vector object while `p1` and `p2` are points.

Regarding transformations, vectors and points behave differently. While for a point, the displacement is applied, it is not for vectors. So

```
p' = T * p = M * p + d
v' = T * v = M * v
```

Here `M` is the 2x2 rotation/mirror matrix part of the transformation and `d` is the displacement vector

The reason why the displacement is not applied to a vector is seen here:

```
v' = T * v
   = T * (p2 - p1)
   = T * p2 - T * p1
   = (M * p2 + d) - (M * p1 + d)
   = M * p2 + d - M * p1 - d
   = M * p2 - M * p1
   = M * (p2 - p1)
```

where the latter simply is:

```
v' = M * v
```

Simple transformations

Simple transformations are represented by DTrans or Trans objects. The first operates with floating-point displacements in units of micrometers while the second one with integer displacements in database units. "DTrans" objects act on "D" type floating-point coordinate shapes (e.g. DBox) while "Trans" objects act on the integer coordinate shapes (e.g. Box).

The basic construction parameters of "DTrans" and "Trans" are:

```
Trans(angle, mirror, displacement)
DTrans(angle, mirror, displacement)
```

"displacement" is a DVector (for DTrans) or a Vector (for Trans).

"angle" is the rotation angle in units of 90 degree and "mirror" is the mirror flag:

angle	mirror = False	mirror = True
0	r0	m0
1	r90	m45
2	r180	m90
3	r270	m135

Complex transformations

Complex transformations in addition to the simple transformations feature scaling (magnification) and arbitrary rotation angles. Other than simple transformations they do not necessarily preserve a grid and rounding is implied. Furthermore they imply a shift is physical scale which renders them difficult to use in physical frameworks (e.g. DRC). Hence their use is discouraged for certain applications.

The basic classes are DCplxTrans for the micrometer-unit (floating-point) version and ICplxTrans for the database-unit (integer) version. The construction parameters are:

```
ICplxTrans(angle, mirror, magnification, displacement)
DCplxTrans(angle, mirror, magnification, displacement)
```

Here, "angle" is the rotation angle in degree (note the difference to "Trans" and "DTrans" where the rotation angle is in units for 90 degree. "magnification" is a factor (1.0 for "no change in scale").

There are two other variants useful for transforming coordinate systems: CplxTrans takes integer-unit objects and converts them to floating-point unit objects. It can be used to convert from database units to micrometer units when configured with a magnification equal to the database unit value:

```
T = CplxTrans(magnification: dbu)
q = T * p
```

The other variant is VCplxTrans which converts floating-point unit objects to integer-unit ones. These objects are generated when inverting "CplxTrans" objects.

2.4. About Expressions

Beside a ruby and Python programming API, KLayout provides support for simple expressions in some places. In particular this feature is employed to generate dynamic strings, for example when deriving the label text for a ruler.

String interpolation

The feature of inserting dynamic content into a string is called interpolation. The Syntax KLayout uses for string interpolation is a dollar character followed by the expression which is evaluated. Simple expressions can be put directly after the dollar character. Others must be put into brackets.

Every dollar expression is evaluated and the expression is substituted by the result string. For example:

String	Evaluates to
`An irrational number: $sqrt(2).`	`An irrational number: 1.4142136`
`1+2 is $(1+2).`	`1+2 is 3.`

String interpolation plays a role where expressions are used to generate dynamic strings. When expressions are used as standalone features (i.e. as parts of a custom layout query - see About Custom Layout Queries), string interpolation is not supported inside string constants, but strings can be built dynamically using the "+" operator.

Basic data types

Expressions use different data types to represent strings or numeric values. The following data types are supported currently:

Type	Examples
Numeric	`1.2` `-0.5e-6`
String	`"abc"` `'x'`
Boolean	`true` `false`
Array	`[1,5,4]`
Undefined (no value)	`nil`

Apart from that, all RBA objects are supported with their methods (see Class Index). For example, that is a valid expression which gives a value of 100:

```
Box.new(-10, 0, 90, 60).width
```

In a boolean context (i.e. the conditional evaluation "condition ? expr1 : expr2"), "nil" and the boolean "false" will render false, while all other values render true. This follows the Ruby convention and in effect, unlike some other languages, a numeric value if 0 is not treated as "false" but as "true"!

Constants

The following constants are defined currently:

Constant	Description
`M_PI`	The mathematical constant 'pi'
`M_E`	The mathematical constant 'e'
`false`	'false' boolean value
`true`	'true' boolean value
`nil`	The 'undefined' value

Operators and precedence

KLayout's expressions support the following operators with the given precedence:

Prec.	Operator	Data types	Result type	Description
1	(...)	Any		Grouping of sub-expressions
2	[...,...]	Any	Array	Array formation
3	!...	Boolean	Boolean	Logical 'not'
3	~...	Numeric	Numeric	Bitwise 'not' (evaluated as 32 bit integers)
3	-...	Numeric	Numeric	Negation
4	...^...	Numeric	Numeric	Bitwise 'xor' (evaluated as 32 bit integers)
4	...&...	Numeric	Numeric	Bitwise 'and' (evaluated as 32 bit integers)
4	...|...	Numeric	Numeric	Bitwise 'or' (evaluated as 32 bit integers)
5	...%...	Numeric	Numeric	Modulo
5	.../...	Numeric	Numeric	Division
5	...*...	Numeric	Numeric	Product
		Numeric*String	String	String multiplication (n times the same string)
6	...-...	Numeric	Numeric	Subtraction
6	...+...	Numeric	Numeric	Addition
		String	string	Concatenation
7	...<<...	Numeric	Numeric	Bit shift to left
7	...>>...	Numeric	Numeric	Bit shift to right
8	...==...	Any	Boolean	Equality
8	...!=...	Any	Boolean	Inequality
8	...<=...	Any	Boolean	Less or equal
8	...<...	Any	Boolean	Less
8	...>=...	Any	Boolean	Greater or equal
8	...>...	Any	Boolean	Greater
8	...~...	String	Boolean	Match with a glob expression
8	...!~...	String	Boolean	Non-match with a glob expression
9	...&&...	Boolean	Boolean	Logical 'and'
9	...||...	Boolean	Boolean	Logical 'or'
10	...?...:...	Boolean?Any:Any	Any	Conditional evaluation

The match operators work on strings. They use the glob pattern notation (as used in the shell for example) and support substring matching with the usual bracket notation. Substrings can be referred to by "$n" later, where n is the nth bracket. For example:

Expression	Result
`"foo" ~ "f*"`	true
`"foo" ~ "bar"`	false
`"foo" !~ "bar"`	true
`"foo" ~ "f(*)"; $1`	"oo"

Method calls

Expressions support all the objects provided by KLayout for the Ruby API. Objects are values inside expressions like integers or strings are. Sometimes, objects can be manipulated with the operators as well (like "box1 + box2"). The most important way to work with objects however are methods.

The dot calls a method on an object. Before the dot an expression must be given which results in an object, or a class name must be given. In the latter case, static methods will be called. After the dot, a valid method name is expected.

Important note: the method names used inside expressions usually is equivalent to the names mentioned in the class documentation. Setter methods like "box_with=" can be used as targets in assignments, i.e.

```
shape.box_width = 20
```

Boolean predicates (like "is_box?") are used **without** the question mark because that is reserved for the decision operator (".. ? .. : .."):

```
shape.is_box
```

Concatenation of expressions

The semicolon separates two expressions. The value of that compound expression is the value of the last one.

Keyword arguments

Most methods support keyword arguments similar to Python. For example you can write:

```
CplxTrans.new(rot = 45.0)
```

This is more explicit than writing the individual arguments and allows giving one argument without having to insert the default values for the previous ones.

Keyword arguments are not supported for the built-in functions such as "sqrt" and a few built-in methods.

Variables

Depending on the context, some variables may be already defined. For example, when used for generating ruler dimension labels, "D" is a predefined variable that is the length of the ruler. See the specific documentation on these variables.

Inside expressions, variables can be defined to store intermediate results for example. To define a variable use the "var" keyword followed by the variable name and an optional initialisation. Assignment of values can be done with the "=" operator. For example, the following expression gives the result 4:

```
var x = 3; x = x + 1; x
```

Special variables

Special variables start with a dollar character. Currently the only special variables available are "$1..9" which is the 1 to 9nth substring match of the last match expression.

Special constants

In the context of a layout, various additional constant expressions are supported:

Distance and area units

A value with a unit is automatically converted to database units. For example, "`0.15 um`" will give 150 if the database unit of the layout is 1 nm. See below for a list of units available. Supported units are:

Unit	Description
`nm`	Nanometers
`um`, `mic`, `micron`	Micrometers
`mm`	Millimeters
`m`	Meters
`nm2`	Square nanometers
`um2`, `mic2`, `micron2`	Square micrometers
`mm2`	Square millimeters
`m2`	Square meters (for very big chips)

Layer index constants

A layer given in the common notation and enclosed in angle brackets is converted to a layer index. For example: "`<16/0>`" will be converted to the layer index of the layer with layer number 16 and datatype 0.

Cell index constants

A cell name enclosed in double angle brackets will be converted to the index of that cell, for example "`<<TOP>>`".

Functions

KLayout's expressions support the following functions:

Function	Data types	Result type	Description
`absolute_file_path(x)`	String	String	Convert a relative file path to an absolute one
`absolute_path(x)`	String	String	Returns the absolute path component of a file specification
`abs(x)`	Numeric	Numeric	Returns the absolute value of a number
`acos(x)`	Numeric	Numeric	Inverse cosine function
`asin(x)`	Numeric	Numeric	Inverse sine function
`atan2(x,y)`	Numeric	Numeric	Inverse tangent of x/y
`atan(x)`	Numeric	Numeric	Inverse tangent function
`basename(x)`	String	String	Returns the basename component of a file specification
`ceil(x)`	Numeric	Numeric	Round up
`combine(x,y)`	String	String	Combines the path components x and y using the system specific separator
`cosh(x)`	Numeric	Numeric	Hyperbolic cosine function
`cos(x)`	Numeric	Numeric	Cosine function
`env(x)`	String	String	Access an environment variable
`error(x)`	String		Raise an error
`exp(x)`	Numeric	Numeric	Exponential function
`extension(x)`	String	String	Returns the extension component of a file specification
`file_exists(x)`	String	Boolean	Returns true if the given file exists

Function	Data types	Result type	Description
`find(s,t)`	String	Numeric	Finds the first occurrence of a t in s and returns the position (where 0 is the first character)
`floor(x)`	Numeric	Numeric	Round down
`gsub(s,x,y)`	String	String	Substitute all occurrences of a x in s by y
`is_array(x)`	Any	Boolean	True if the argument is an array
`is_dir(x)`	String	Boolean	Returns true if the given path is a directory
`is_nil(x)`	Any	Boolean	True if the argument is undefined
`is_numeric(x)`	Any	Boolean	True if the argument is numeric
`is_string(x)`	Any	Boolean	True if the argument is a string
`item(a,i)`	Array	Any	Access a certain item of an array
`join(a,s)`	Array, String	String	Join all array members in a into a string using the separator s
`len(x)`	String	Numeric	Return the length of a string
`log10(x)`	Numeric	Numeric	Base 10 logarithm function
`log(x)`	Numeric	Numeric	Natural logarithm function
`max(a,b ...)`	Numeric	Numeric	Maximum of the given arguments
`min(a,b ...)`	Numeric	Numeric	Minimum of the given arguments
`path(x)`	String	String	Returns the path component of a file specification
`pow(x,y)`	Numeric	Numeric	Power function (x to the power of y)
`rfind(s,t)`	String	Numeric	Finds the last occurrence of a t in s and returns the position (where 0 is the first character)
`round(x)`	Numeric	Numeric	Round up or down
`sinh(x)`	Numeric	Numeric	Hyperbolic sine function
`sin(x)`	Numeric	Numeric	Sine function
`split(t,s)`	String	Array	Splits t into elements using the separator s
`sprintf(f,a ...)`	String, Any	String	Implement of 'C' sprintf. Provides not all features but the most commonly used ones (precision, field width, alignment, zero padding, 'e', 'g', 'f', 'd', 'x', 'u' and 's' formats)
`sqrt(x)`	Numeric	Numeric	Square root
`substr(t,f[,l])`	String	String	Return a substring of t (starting from position f with length l). 'l' is optional. If omitted, the tail of the string is returned.
`sub(s,x,y)`	String	String	Substitute the first occurrence of a x in s by y
`tanh(x)`	Numeric	Numeric	Hyperbolic tangent function
`tan(x)`	Numeric	Numeric	Tangent function
`to_f(x)`	Any	Numeric	Convert argument to numeric if possible
`to_i(x)`	Any	Numeric (integer)	Convert argument to numeric (32 bit integer)
`to_s(x)`	Any	String	Convert argument to string

2.5. About Variant Notation

KLayout employs a certain notation to enter variant types. Variant types are data items which are either numerical or text information and are used specifically for property names and values. In that case, the type of information is important, and a simple edit box won't be sufficient to enter the information. For example, "1" could either be a text "1" or the number 1. To solve that issue, KLayout uses the following notation:

Notation	Description	Example
#...	An integer value	`#17`
##...	A floating-point value	`##0.5`

'...' or "..."	A text string which can contain any character. You can use the backslash character to escape quotes if you want to use quotes in your text string. To use the backslash character inside the text, use a double backslash (\\).	`'A text'`
		`'A \'quoted text\''`
		`'A single \\ character'`

A word (letters, digits and the underscore)	Taken as a text string	`NAME`

2.6. About LEF/DEF Import

KLayout supports import of LEF and DEF files. Because LEF and DEF import is substantially different from a single-file reader, this functionality is wrapped in an import feature rather than a standard file format reader. For example, reading a DEF file without accessing the library LEF files does not make much sense. Therefore the import feature requires additional information beyond the simple file name, specifically the list of LEF files to load additionally and the order in which to do so.

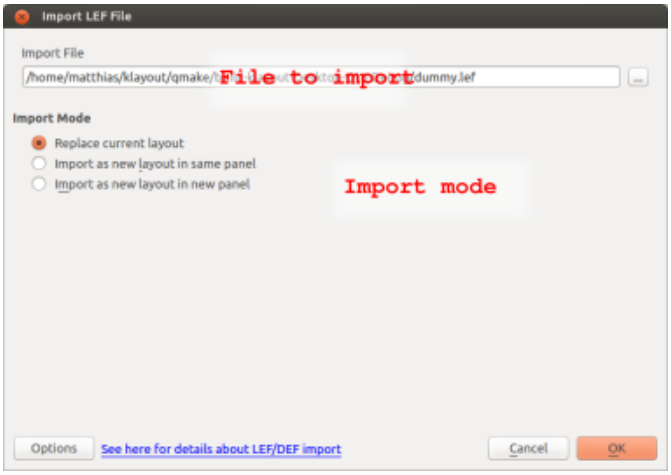
You can find the import feature in the "File/Import" submenu. Choose "LEF" to import a single LEF file (plus potentially more LEF files from the technology specific settings) or "DEF/LEF" to import a DEF file plus additional LEF files.

In the import dialog, the files to import are selected. The top edit box specifies the main file to import (the LEF file in the LEF import case or the DEF file in the DEF import case. Use the "..." button to browse for the file. In the DEF import case, additional LEF file can be specified which are imported before the DEF file is read. You can add LEF files to the list using the "+" button and remove the selected LEF files using the "x" button. The order of LEF files matters - the technology LEF file (if there is one) should be read first. The LEF files are read in the order they appear in the list. You can move entries up using the "up" button and down using the "down" button.

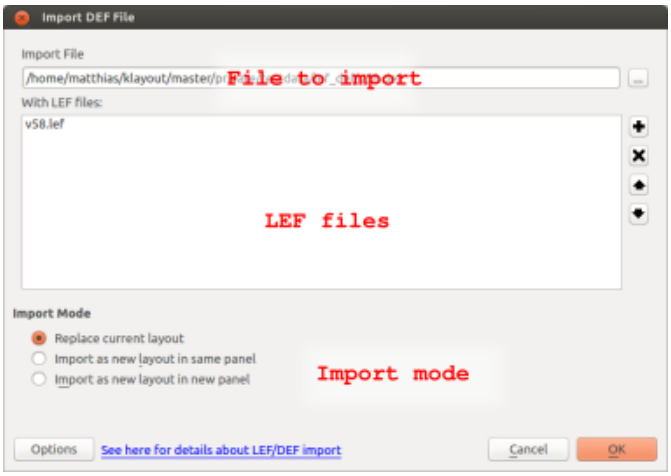
If you browse for a DEF file, KLayout will automatically fill the list of LEF files with the LEF files found at the location of the DEF file. LEF files are searched for relative to the DEF file path. Unless an absolute path is specified for the LEF file, KLayout will look relative to the path of the DEF file.

In both LEF or DEF import case you have the choice to read the layout into the existing view (add to the current layout or overwrite the current one) or open a new view using the "Import Mode" options.

The following image shows the LEF import dialog:

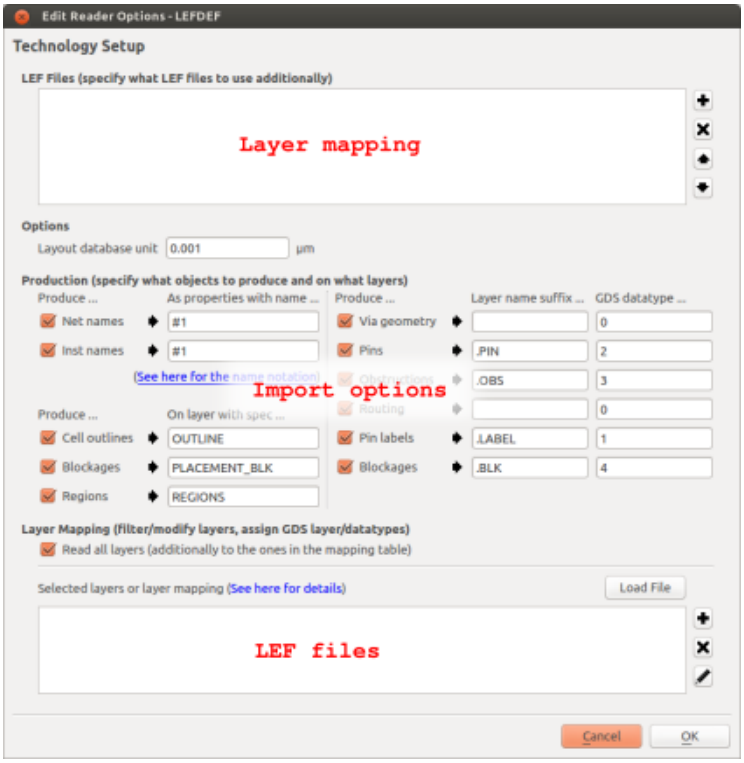


And this one is the DEF import dialog:



LEF and DEF import can be configured in many ways. The configuration of the LEF/DEF import feature is attached to a technology, so there can be individual configurations per technology. For a description of the technology feature, see About Technology Management. The import feature uses the current technology. The current technology can be selected from the main toolbar's technology selector if specific technology settings exist. Otherwise, the default technology will be used.

You can edit the import settings using the "Options" button from the import dialogs. This will open a dialog for editing the settings attached to the selected technology. The settings basically consist of three parts: the layer mapping, the LEF files which are always read before the LEF files specified per DEF file and various import settings. The following image shows the LEF/DEF import options dialog:



Layer Mapping

By default, no layer mapping is specified. Layer mapping can be employed to confine input to certain layers or layer/purpose pairs and to specify mapping of LEF layers/purposes to GDS layer/datatypes.

In the Options part you will learn how the DEF/LEF importer generates layers. Basically, layers are generated from the LEF/DEF layer name plus some suffix describing the purpose. For example, pin geometry will be put to "Metal1.PIN" for "Metal1" geometry if the suffix for the "pin" purpose is set to ".PIN". In addition, the LEF/DEF reader will already assign some GDS layer/datatype, the layer is either the position of the layer in the layer list if there is a technology LEF file or a consecutive number based on the alphabetic order of the layer names. The datatype can be specified for each purpose.

However, that may not be sufficient to convert a DEF file to a certain GDS representation. Usually there is a layer mapping table, for example "Metal1.PIN" should be put to GDS layer 17, datatype 6. In order to facilitate such a use case, KLayout allows specification of a layer mapping table. The basis for the table is the layer name plus the suffix defined for the particular purpose. In the previous example, the mapping would be:

```
Metal1.OBS : 100/22
```

You can disable all other layer/purpose pairs by unchecking the "Read all layers" option at the top of the layer mapping table. If this option is enabled, KLayout will generate the GDS layers using the scheme described above for all layers it does not find in the mapping table.

LEF Files

You can specify a list of LEF files to load before any other LEF files. This feature is intended to allow specification of a technology LEF file containing the layer definitions or any standard library which should be present for every macro built on that technology. You can add and delete LEF files using the "+" and "x" button respectively. You can move files up and down using the "up" and "down" arrow buttons. The LEF files are read in the given order, so the technology LEF file must be the first in the list.

Options

On the left side of the option panel, two basic options are provided:

- **Produce net names:** Check this option to assign user properties with the net name to the net shapes in DEF files. The user property name used for that purpose can be specified in the edit box below the check box. Use KLayout's variant notation (see About Variant Notation) to specify value and type of the property name.
- **Produce inst names:** Check this option to assign user properties with the instance name to the component instances created by DEF import. The user property name used for the instance name can be specified in the edit box below the check box. Use KLayout's variant notation (see About Variant Notation) to specify value and type of the property name.
- **Produce cell outlines:** If this option is checked, outline shapes are produced for the macros and the design (for DEF import). The layer to be used can be specified in the edit box below. You can use KLayout's usual layer specification notation, i.e. "OUTLINE" for a named layer without GDS layer/datatype value, "10/0" for GDS layer 10, datatype 0 without a name or "OUTLINE (10/0)" for a combination of both. The outline layer is subject to layer mapping as well, so the layer map can be used alternatively to assign GDS layer/datatype numbers.
- **Produce blockages:** If this option is checked, placement blockages are produced as polygons on the layer given right of the check box. Use KLayout's layer notation to specify the layer (see "Produce cell outlines").
- **Produce regions:** If this option is checked, regions are produced as polygons on the layer given right of the check box. Use KLayout's layer notation to specify the layer (see "Produce cell outlines").

On the right side, the default layer generation for various purposes can be configured. In all cases, a layer suffix can be set which is just added to the layer name and a default GDS datatype can be set. Please note, that the GDS datatype may be overruled by a layer map if one is set. All contributions can be disabled individually.

The purposes available are:

- **Via geometry:** generated for shapes making up a via.
- **Pins:** generated for shapes making up a pin.
- **Obstructions:** generated for obstruction area markers.

- **Routing:** generated for routing geometry.
- **Pin labels:** generated for pin labels.
- **Blockages:** generated for (component) blockages.

Note: if the suffix of two purposes is identical, the default GDS datatype should be identical as well. Otherwise it is not defined which layer will be generated. Vice versa, the GDS datatypes should be different for different layer suffixes.

2.7. Connectivity

Use the connectivity page to specify the conductor layers and their connections. On a conductor layer, all touching or overlapping shapes are connected. A connection is made between conductor layer when the shapes of the two conductor layers overlap. Optionally a via layer can be specified which must be present to make a connection between the two conductor layer.

To specify a conductor layer

- Use "layer/datatype" notation to specify explicit GDS layers and datatypes.
- Enter the layer name to specify either a named layer or a symbolic layer. Symbolic layers must be defined in the symbol table (see Symbolic Connectivity Layers) and can be computed from boolean expressions.
- Instead of using a symbolic layer, enter a expression directly without defining a symbol (see Symbolic Connectivity Layers). Inside the expressions
    - Use "layer/datatype" notation to specify an original layer with explicit GDS layers and datatypes.
    - Use the name to specify a named original layer or to refer to a different symbolic layer defined in a symbol entry.
    - Use the operators '+', '*', '-' and '^' to specify logical OR, AND, NOT and XOR respectively. The precedence of evaluation is '^' and '*' before '+' and '-'.
    - Use round brackets to group expressions.

By creating conductor layers with boolean expressions, it is possible for example to separate an active area layer of a CMOS transistor into source and drain regions by subtracting the gate poly. Symbolic layers are useful to use "speaking" names for layers instead of the numeric layer/datatype specification. Please note, that the net tracer is considerably slower when using boolean expressions.

2.8. The 2.5d View

The "2.5d view" offers a semi-3d view of the layout. It's not a full 3d view as the layers are only extruded vertically into layers with a certain thickness. The view cannot model process topology, but it can visualize wiring congestions in a three-dimensional space or the vertical relative dimensions of features of the process stack.

Currently, the performance is limited, a rough number for a practical limit is around 100k polygons. The 2.5d view is only available, if KLayout was compiled with OpenGL support.

In order to use the tool, you will need a script generating the material stack. Such a script is a variant of a DRC script (see Design Rule Check (DRC)). The DRC language is used to import or generate polygon layers which are then extruded and placed on a certain z level.

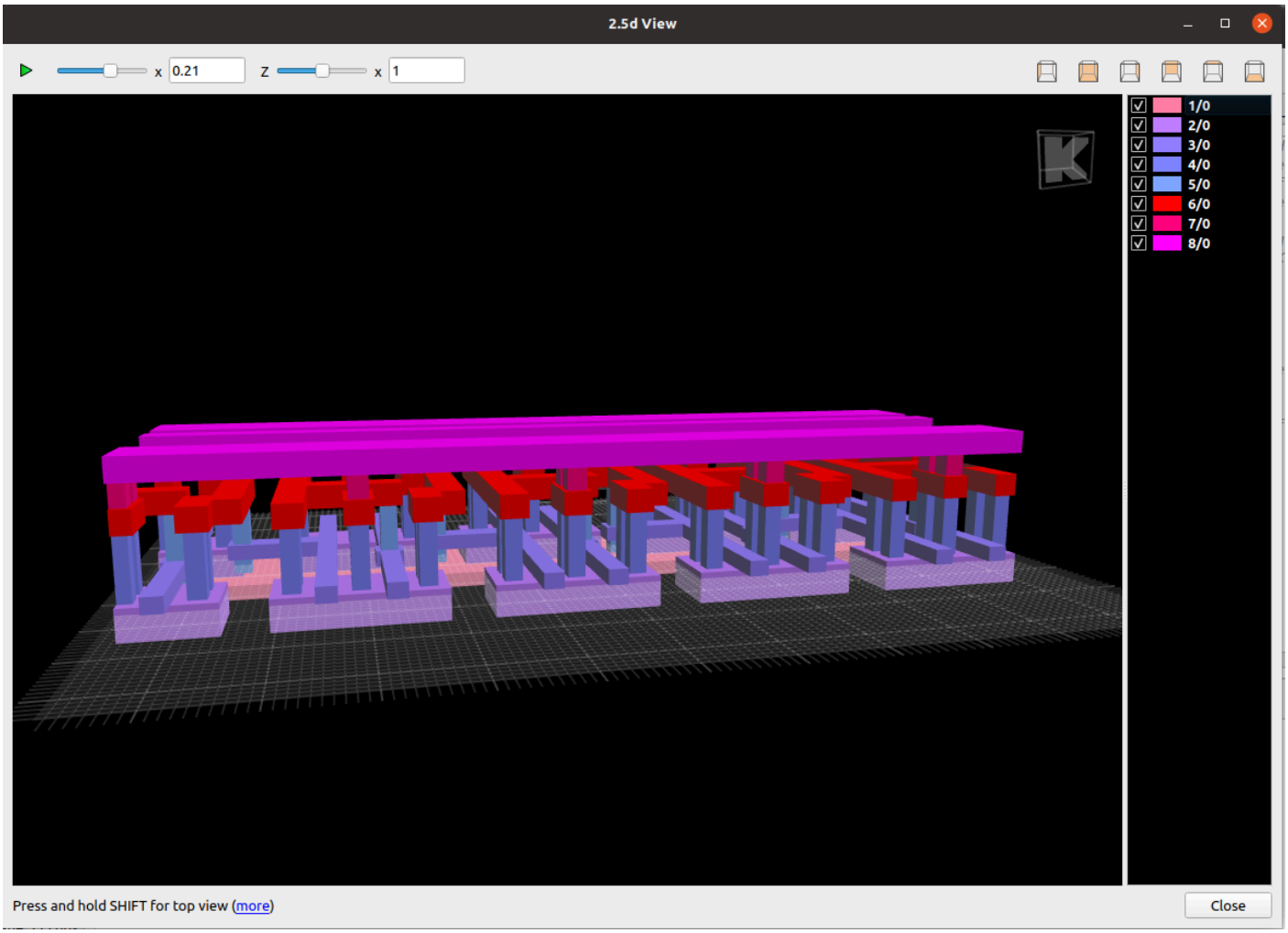
To create a new script, use "Tools/2.5d View/New 2.5d Script". This will create a new script in the macro editor.

A simple script is this one. It takes two layers - 1/0 and 2/0 - and extrudes then in a stacked fashion, the first with 200nm thickness and the second one with 300nm:

```
z(input(1, 0), zstart: 0.1.um, height: 200.nm)  # extrudes layer 1/0 to a height of 200nm starting at z=100nm
z(input(2, 0), height: 300.nm)                  # adds layer 2/0 for the next 300nm
```

To run the script, use the "Run" button from the macro IDE or pick the script from the script list in the "Tools/2.5d View" menu. If your script is not shown in that menu, check if it is configured to be bound to a menu item.

After the script was executed, the 2.5d window is displayed. If you closed that window, you can re-open it with "Tools/2.5d View/Open Window". The window will show the layout section visible in the layout view. To refresh the scene - also after changing the script - either run the script again from the macro IDE or use the green "re-run" button in the upper left corner of the 2.5d view window.



2.5d Script Anatomy

As mentioned, a 2.5d script is a variant of a DRC script. You can basically use all features of DRC, specifically boolean operations. Some practical restrictions exist:

- You should not use external sources ("source" statement) as the 2.5d view is related to the loaded layout
- Report generation or "output" statements are permitted, but do not make much sense in the context of 2.5d view scripts.

2.5d scripts utilizes the DRC language with these two additional functions:

- z(layer [, options])

  Extrudes the given layer. "layer" is a DRC layer (polygon, edge or even edge pair). "options" declare the z extrusion and display parameters.

- zz([options]) { block }

  Declares a material group which combines multiple "z" statements under a single display group. This allows generating 3d material geometries which are more than a single extruded plane. The display parameters then are specified within "zz" for all "z" calls inside the block.

"z" Function (plane extrusion)

The layer argument of the function is a DRC layer which is rendered as an extruded sheet. Further arguments control the height, z location and colors. When used inside the "zz" block, the color options of the "z" calls are ignored and taken from "zz" instead.

Options for this function are:

- **zstart**: specifies the bottom coordinate of the extruded sheet. If this option is not given, the top coordinate of the previous "z" statement is used.
- **zstop**: specifies the top coordinate of the extruded sheet. Alternatively you can use "height".
- **height**: specifies the extrusion height. Alternatively you can use "zstop".
- **color**: specifies the color to use as a 24 bit hex RGB triplet (use "0xrrggbb" to specify the color similar to the HTML notation "#rrggbb"). A color specification gives a single color with not differentiation of frame and wall colors.
- **frame**: specifies the frame color to use as a 24 bit hex RGB triplet. If only a frame color is specified, the geometry will be rendered as wire frame only.
- **fill**: specifies the fill (wall) color to use as a 24 bit hex RGB triplet. This allows specifying a different color for wall and frame when used with "frame".
- **like**: specifies to use the same colors than used for some layer in the layout view. If the layer is an original layer (i.e. taken from "input"), "like" defaults to the original layer's source. If given, "like" needs to be a string representation of the layer source (e.g. "7/0" for layer 7, datatype 0).
- **name**: gives the material a name for displaying in the material list.

Examples for the extrusion options:

```
z(layer, 0.1 .. 0.2)               extrude layer to z = 0.1 to 0.2 um
z(layer, zstart: 0.1, zstop: 0.2)  same as above
z(layer, zstart: 0.1, height: 0.1) same as above, but with height instead of zstop
z(layer, height: 200.nm)           extrude layer from last z position with a height of 200nm
```

Examples for display options:

```
z(..., color: 0xff0000)            use bright red for the material color (RGB)
z(..., frame: 0xff0000)            use bright red for the frame color (combine with "fill" for the fill
 color)
z(..., fill: 0x00ff00)             use bright green for the fill color along (combine with "frame" for the
 frame color)
z(..., like: "7/0")                borrow style from layout view's style for layer "7/0"
z(..., name: "M1")                 assigns a name to show for the material
```

"zz" Function (material groups)

The "zz" function forms a display group which clusters multiple "z" calls. The basic usage is with a block containing the "z" calls. As DRC scripts are Ruby, the notation for the block is either "do .. end" or curly brackets immediately after the "zz" call:

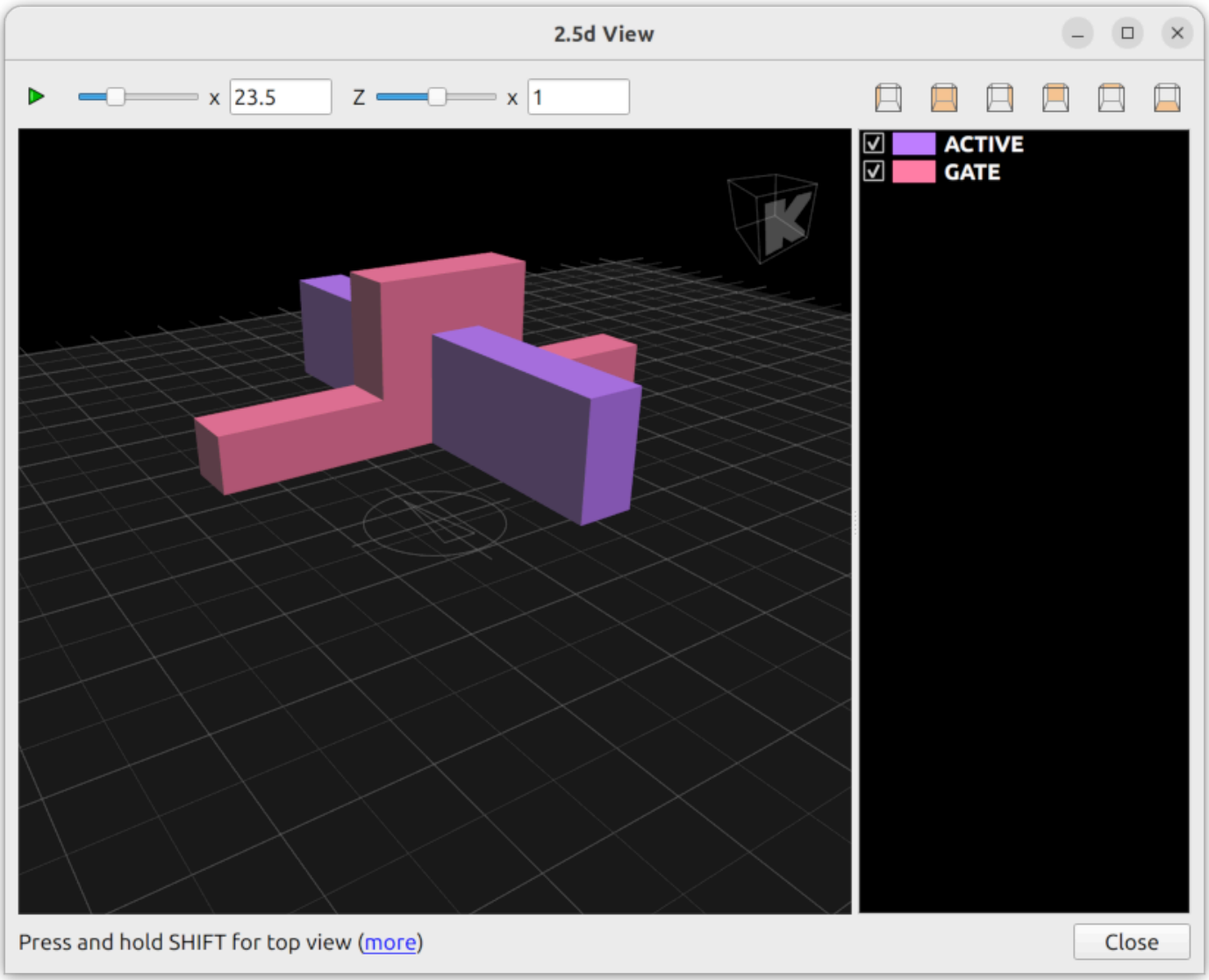
```
zz( display options ... ) do
  z(layer1, extrusion options ... )
  z(layer2, extrusion options ... )
  ...
end
```

The "z" calls do not need to have colors or other display options as they are taken from "zz".

Material groups allow forming more complex, stacked geometries. Here is an example forming a simple FinFET geometry using boolean and a sizing operation:

```
gate = input(1, 0)
active = input(2, 0)

z(active, zstart: 0, height: 20.nm, name: "ACTIVE")

zz(name: "GATE", like: "1/0") do

  active_sized = active.sized(10.nm)
  gate_over_active_sized = active_sized & gate

  z(gate - active, zstart: 0, height: 10.nm)
  z(gate_over_active_sized - active, height: 10.nm)
  z(gate_over_active_sized, height: 10.nm)

end
```

Which renders this result:



Navigating the 2.5d View

The navigation is based on the movement of the camera while the scene is formed by the extruded layout. The scene can be scaled to provide zoom features. Scaling and rotation is relative to the pivot point which is indicated by the compass icon on the ground plane.

This is a short list of the navigation controls which act on the camera:

- Dragging with the right mouse button down: change azimuth and elevation angle
- Dragging with the middle mouse button down: move the pivot up and down or left and right
- Mouse wheel: moves the pivot forward and backward
- Control key + mouse wheel: magnify or shrink the layout
- Press and hold shift key: switch to top level view (see below)
- Up/down keys: move the pivot forward or backward

- Left/right keys: move the pivot to the left or the right
- Control + up/down keys: change the elevation angle
- Control + left/right keys: change the azimuth angle

In top level view, the navigation is slightly different:

- Dragging with the right mouse button down: change azimuth angle
- mouse wheel: magnify or shrink the layout
- Up/down/left/right keys: move the pivot on the horizontal plane

Note: if the Shift key does not switch to top level view, click into the scene view once.

Colors in the 2.5d View

While the 2.5d view window is open, the layout view is still active. Layer colors are applied also to the 2.5d view. Changing the fill color will change the 2.5d view's face color. The frame color will be applied to the wire frame. If a hollow stipple is selected, only the wire frame is shown. If a layer is made invisible in the layout view, the corresponding blocks will also be made invisible in the 2.5d view.

Other Controls

The left zoom slider changes the overall scale factor. The right slider only changes the z (height) axis zoom factor. This is useful as in many cases, the real height profile will result in a rather flat arrangement. Increasing the z zoom factor will exaggerate the vertical axis hence making height variations more pronounced.

The edit boxes next to the scale sliders allow entering the scale factors manually.

The button bar at the top right side holds the view presets. Use them to reset the window to front view, top view etc.

Material Visibility

Using the check boxes from the material view right of the scene view you can disable materials, so they are no longer rendered. From the material list's context menu, you can hide or show all materials or just the selected ones.

2.9. Symbolic Connectivity Layers

Use the symbol table to specify derived layers and to assign names to layer/datatype combinations. A symbolic layer must have a name which can be used in the connectivity table instead of the original layers. In addition, an expression must be specified that defines the contents of the layer.

Inside the expressions

- Use "layer/datatype" notation to specify an original layer with explicit GDS layers and datatypes.
- Use the name to specify a named original layer or to refer to a symbolic layer defined in another entry.
- Use the operators '+', '*', '-' and '^' to specify logical OR, AND, NOT and XOR respectively. The precedence of evaluation is '^' and '*' before '+' and '-'.
- Use round brackets to group expressions.

Examples:

17	Abbreviation for GDS layer 17, datatype 0
2/0*17/0	Logical AND between layer 2 and 17, both datatype 0
2*(5+7)	Logical AND between layer 2 and the logical OR combination of 5 and 7
ACTIVE-POLY	Logical NOT between the symbolic layer ACTIVE (defined in another entry) and POLY (also defined in another entry).

2.10. About Layer Sources

KLayout implements a concept of "layer views". The layer list is made up of such layer views. A "view" is basically a specification of what is shown how. The "how" part is given by the colors, stipples, styles etc. The "what" part is given by the source specification.

The most important part of the source specification is the layer number or name. But the source specification is much more powerful. Basically the source specification offers the following capabilities:

- Specify the layer where the shapes are taken from, either by layer and datatype or name
- Transform the layout before it is drawn
- Property filter: draw only shapes whose user properties match a given expression
- Override the hierarchy levels on which the shapes are drawn

Specifying the source layer

The source layer specifies from which actual data layer to take the drawn shapes from. The most simple form of a source specification is "layer/datatype" (i.e. "5/0") or the layer name, if an OASIS layer name (or a named layer in general) is present. This specification can be enhanced by a layout index. The first layout loaded in the panel is referred to which "@1" or by omitting this specification. The source specification "10/5@2" therefore refers to layer 10, datatype 5 of the second layout loaded in the panel.

Source specifications can be wildcarded. That means, either layer, datatype or layout index can be specified by "*". In this case, such a layer view must be contained in a group and the group parent must provide the missing specifications. For example, if a layer is specified "10/*" and the parent is specified "*/5", the effective layer looked for will be "10/5". Unlike the behaviour for the display styles, the children override (or specialize) the parent's definition in the case of the source specification.

For more information refer to Removing And Adding Layers To The Layer Set.

Transforming the layout

A geometrical transformation is specified by appending a transformation in round brackets to the layer/datatype source specification.

For example, "(r90)" specifies a rotation by 90 degree counter-clockwise. "(0,100.0 m45 *0.5)" will shrink the layout to half the size, flip at the 45 degree-axis (swap x and y axes) and finally shift the layout by 100 micron upwards.

Transformations accumulate over the layer hierarchy. This means, that if a layer is transformed and the layer is inside a group whose representative specifies a transformation as well, the resulting transformation is the combination of the layer's transformation (first applied) and the group representative's transformation.

Multiple transformations can be present. In this case, the layout is shown in multiple instances.

For more information refer to Transforming Views And Property Selectors.

Property filters

It is possible to specify a property filter. A property filter specifies an expression and only shapes for which that expression applies are shown. The expression operates on user properties and the syntax allows comparison of properties with a given key against a given value. Boolean operators are available. That way, complex expressions can be created.

The property filter is specified in square brackets. For example:

```
10/5 [#43==X]
```

With this source specification, the layer will show all shapes from layer 10, datatype 5 which have a user property with number 43 and value string "X".

For more information refer to Transforming Views And Property Selectors.

Overriding the hierarchy levels

By default, only the hierarchy levels that are selected in the hierarchy level selection boxes are shown, i.e. if levels 0 to 1 are selected, just the top level shapes and instances are shown. This selection can be modified for certain layers or layer groups. To specify a different hierarchy selection for a certain layer, use an optional source specification element, the hierarchy level selector.

For example:

`#*`	Display all hierarchy levels
`#0..1`	Display top level only
`#..5`	Override upper level with 5
`#2..`	Override lower level with 2
`#..*`	Override upper level setting by "all levels"

For more information refer to Specifying Explicit Hierarchy Levels For One Layer Or A Layer Group.

2.11. About Macro Development

Basics

KLayout supports macro programming with the Ruby or Python language. Macros are basically Ruby or Python scripts that are executed by the integrated interpreter. In order to enable macro programming, the program has to be built with support by either one of those languages.

As a special kind of macros, DRC and LVS scripts are available for editing and debugging too. These scripts are basically Ruby scripts but run in a customized environment so that specific functions are provided. For more details on these kind of scripts see Design Rule Check (DRC) and Layout vs. Schematic (LVS).

In a simple scenario, scripts can be stored in simple text files and loaded by KLayout using the "-rm" or "-r" command-line option. "-rm" will run a script and then execute the normal application. "-r" will run a script and then exit the application.

In addition, KLayout supports special macro files with the suffix ".lym". Those files are XML files that store the macro code along with additional information, for example the description text, the interpreter language and certain flags. These flags tell KLayout to run the macro automatically when starting up for example. In addition, KLayout can present such macros in the "Macros" menu without having to register a specific menu extension.

Macros can be technology specific. This means, they are packaged with a technology and are associated with the technology in the user interface: if they provide a menu item, this item will only become visible when the respective technology is active.

KLayout also offers an integrated development environment (IDE) that allows editing and debugging of Ruby and Python scripts. It offers a simple debugger with the ability to set breakpoints and to interact with the current context while in a breakpoint. An interactive "console" allows entering and evaluating of expressions. This feature is available also when execution has stopped in a breakpoint, so the console can be used to evaluate or modify variables in the current context. Watch expressions are supported as well: a series of expressions can be configured which is evaluated and displayed in a breakpoint. With Python scripts, the local context can be inspected in the "Inspector" window: the variables available in the local context are listed with their values.

While the debugger is open, execution will slow down somewhat and undesired interactions may happen - specifically when developing UI components. To mitigate this problem, debugging can be disabled in the macro IDE.

KLayout defines certain folders which it scans for macro files. Each directory is scanned recursively. It can contain subdirectories with more macros and can also contain support files such as images or additional Ruby or Python modules. That allows organizing macros in modules where each module contains the root module files and supporting files.

KLayout looks for macros in the following places:

- The "macros" or "pymacros" folders in the installation path. The installation path is where the KLayout binary resides. KLayout cannot modify macros that are stored in that path. This is the "global" repository. Usually that repository is shared between all users. An administrator can use this location to install macros globally.
- The "macros" or "pymacros" folders in KLayout's user specific application folder. On Unix that is "~/.klayout/macros". This is the "local" repository. Any user can store his or her own macros here.
- Plain Ruby or Python files can be kept inside "ruby" and "python" directories next to "macros" and "pymacros". In contrast to "macros" and "pymacros", the locations of "ruby" and "python" paths are added to the Ruby or Python search paths. This makes those folders useful for keeping plain Ruby or Python libraries. Generic ".lym" files cannot reside there and those locations are not scanned for autorun macros.
- DRC and LVS scripts are kept inside "drc" and "lvs" folders respectively.
- In addition, further repositories can be given on the command line with the "-j" option. This allows adding development repositories which are under configuration management and contain the latest code for the macros. Those repositories are called "project" repositories.
- Technology folders: each technology folder can carry a "macros" or "pymacros" subfolder where technology-specific macros are kept. See About Technology Management for details about technologies.
- Macros can be kept in packages and installed from a remote repository. See About Packages for details about packages.

The macro IDE will store Python and Ruby macros separately in "macros" and "pymacros" for easier management, but technically these folders are equivalent. KLayout will automatically derive the interpreter from the context stored in ".lym" files or the file extension (".rb" or ".py").

You can also add custom places using the "Add Location" entry from the repository browser's context menu (right mouse click). These locations will be added to the tree. That setting is stored in the setup, so it will be available when KLayout is started again. You can remove places again by using "Remove Location" from the macro tree's context menu.

KLayout will scan these places for files and present them in the macro development IDE's repository browser. You can edit the files and their properties and run the macros from the IDE. KLayout's IDE supports these file types:

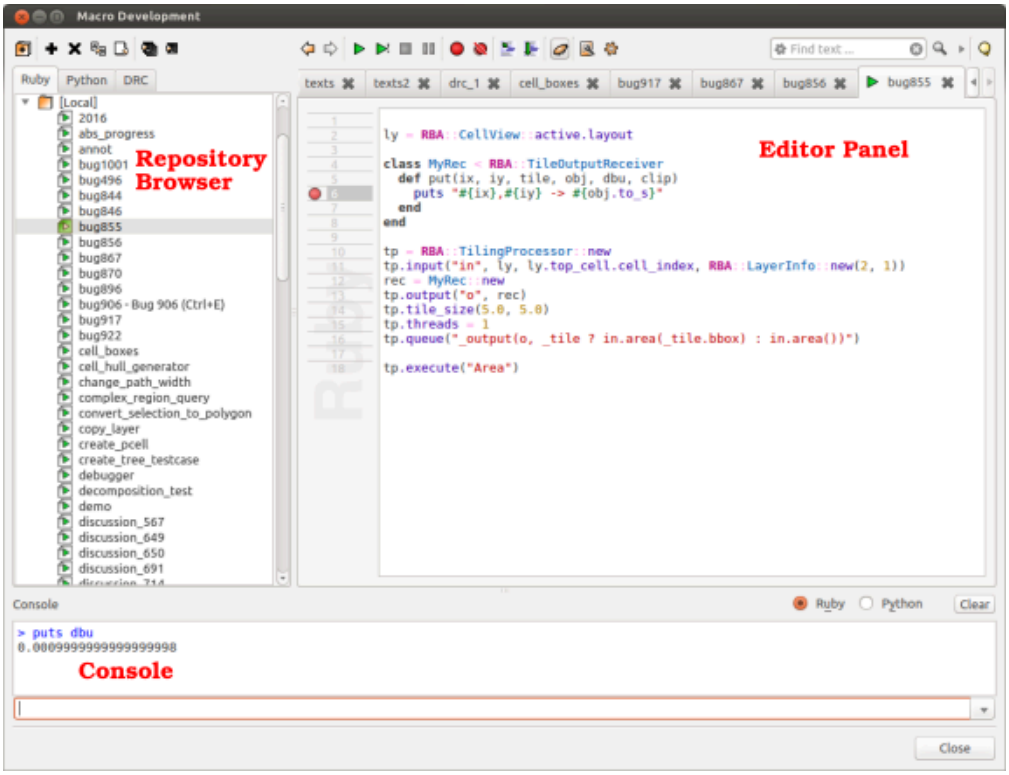
- ".lym": generic KLayout macros.
- ".rb": plain Ruby files. Such files can be imported into KLayout macros by Ruby's "require" or "load" method.
- ".py": plain Python files. Such files can be imported into KLayout macros by Pythons's "import" method.
- ".txt": text files for documentation for example.

Other files can be stored inside the folders (for example Qt ".ui" files), but they won't be shown in the IDE.

The macro development IDE

The macro development IDE consists of three parts: the repository browser, the editor panel and the console. The repository browser allows browsing of the macro repositories and to add new files, to move files, to delete files, to import plain ruby files as KLayout macros and to rename files. In the editor panel, the macro or other files can be edited, breakpoints can be set and the macro can be run. In the console, Ruby expressions can be entered and evaluated. This also works while KLayout is in a breakpoint. That feature evaluating of expressions in the current context and to modify variables. The console also shows the output of the Ruby scripts when they are run in the context of the IDE.

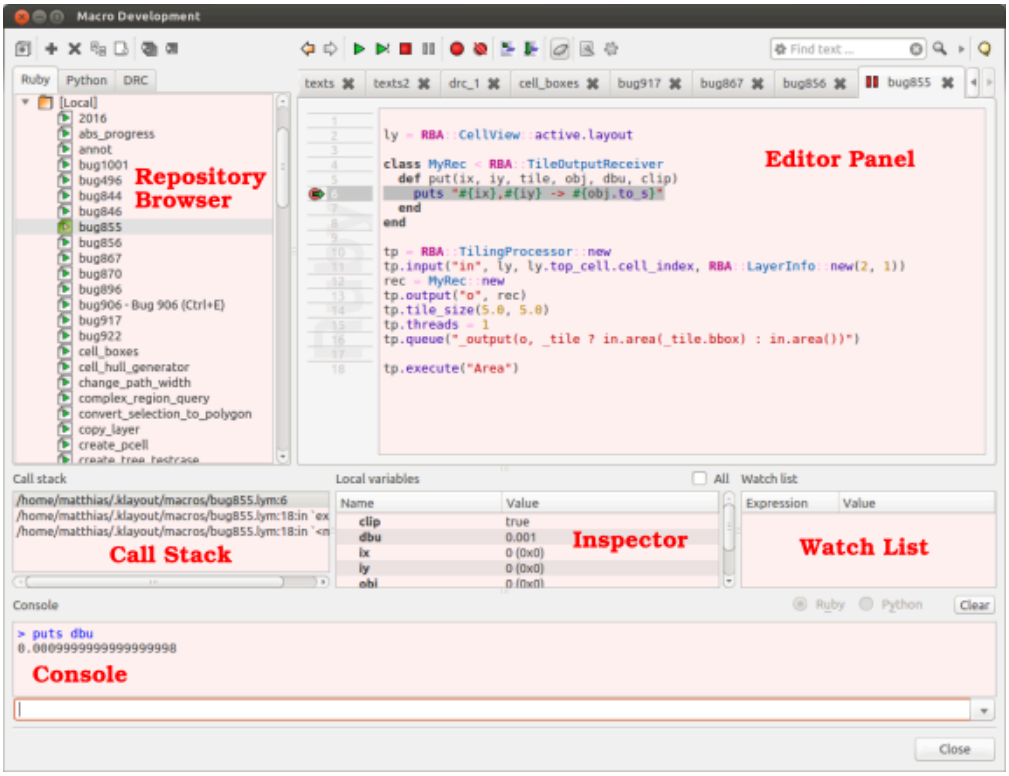
The following image shows the IDE with the three main parts:



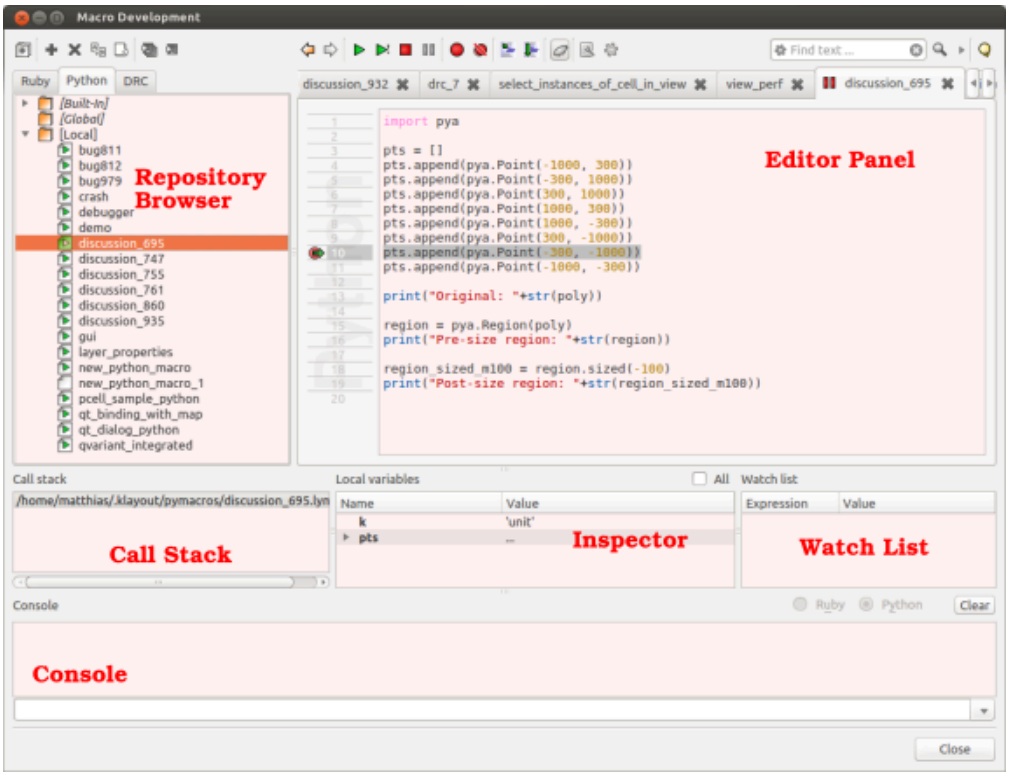
When the debugger enters a breakpoint, the IDE also shows the call stack. By clicking on an entry in the call stack panel, the editor will open the respective file and position the cursor at that line. A watch window shows the watch expressions with their evaluated values.

Watch expressions are evaluated every time a breakpoint it hit. They can be managed using the "Add", "Edit", "Delete" and "Clear" functions from the context menu (right click on the watch list). For Python scripts, the local variables of the currently selected stack context can be browsed in the "Inspector" window in the middle. Currently this feature is not available for Ruby scripts.

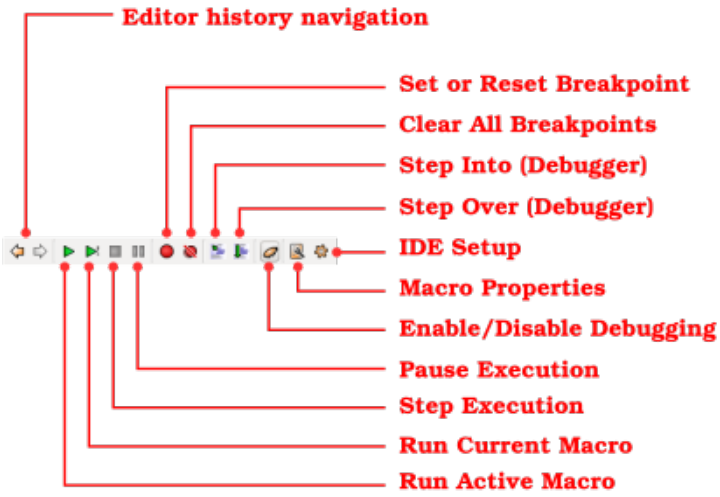
The following image shows the IDE in breakpoint mode for Ruby scripts:



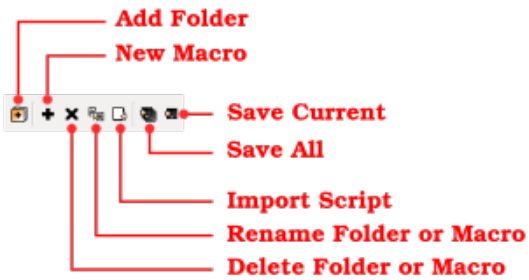
The following image shows the IDE in breakpoint mode for Python scripts:



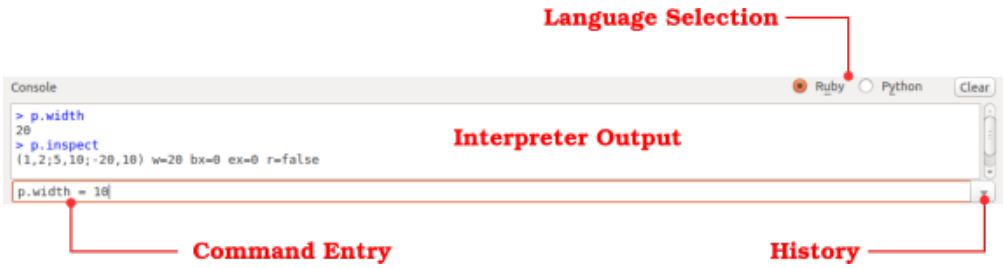
The editor panel features a toolbar with several tool buttons. The following image shows the toolbar and the buttons with their function:



Also the repository browser features a toolbar with more tool buttons. The following image shows that toolbar and the buttons with their function:



Finally the console panel allows interactive executing of Ruby or Python expressions. The following image shows the console elements and their function:



Basic tasks

Creating a new macro

To create a new macro, first open the tab to select a language you want to use. Choose the branch in the repository browser where you want to create the macro. Press the "+" button above the repository browser. A dialog will come up in which you can select a template. A template is basically the initial content of the macro plus some default settings. After the macro is created, the new entry is highlighted and the name can be changed to the desired new name.

Editing the macro

After the macro has been created, it is shown in the editor window. Other macros or files can be opened in the editor by double-clicking at their name. Depending on the type of file, the editor provides some basic syntax highlighting. Each file is opened in a separate tab. The tabs can be closed by clicking at the "x" button in the tab.

To save the current edits to the files, press the "Save all files" or the "Save current file" button in the repository browser.

Search and search & replace is available in the editor panel in the search box. The search a text, enter the text in the search box and hit Return. Use the "Find next" button to find the next occurrence of the text. Click on the "RE" button to enable regular expressions in the search function. To enable "Replace" mode, open the replace text edit box by clicking on the little arrow right to the search tools. Enter the text to replace the search text into the edit box and use "Replace all" to replace all occurrences or use "Replace and search" to replace the current occurrence and highlight the next one.

Running the macro

To run the macro, press the "Run current script" or the "Run script from the current tab" button in the editor panel. "Run current" will run the script that was run the last time, irregardless if the script is the one currently shown in the editor panel. That way, it is possible to run the same script while editing support files for example. "Run script from the current tab" will always run the script from the current tab.

When the macro runs, output will be sent to the console below the editor panel. Breakpoints can be set or reset with the "Set breakpoint" button or the F9 key in the editor panel. Go to the line where you want to change the breakpoint and use "Set breakpoint" to set or reset the breakpoint on the current line. All breakpoints can be cleared with the "Clear all breakpoints" button.

While the macro is running or in a breakpoint, the execution can be aborted with the "Stop script" button. Execution can be interrupted with the "Pause script" button. In that case, KLayout will interrupt the script execution at the current position.

When in a breakpoint, single-line step-by-step is supported with the "Step into" button or the F11 key. Stepping over a block or procedure call is supported with the "Step over" button or the F10 key. Please note that "Step over" will also step over blocks.

While KLayout has stopped the execution of the script, the console can be used to examine variables in the current context or to modify their values. Ruby expressions entered into the console are simply executed in the current context.

When the macro editor window is open, macros will be run under debugger control. This will considerably slow down macro execution and create issues when you develop macros that integrate deeply into the system (for example macros filtering Qt events from the application). To avoid issues, debugging can be disabled by selecting or unselecting the "Enable debugging" button in the editor's toolbar. If the button is checked (shown pressed), debuggin is enabled and breakpoints can be used. If debugging is disabled, breakpoints will be ignored.

Working with the repository browser

The repository browser provides a tab for each category of macros. "DRC" is a special category which provides macros that perform a DRC. Such macros are basically Ruby scripts that are executed in the context of a DRC engine and hence supply different functionality. Still, they execute in the usual context and breakpoints, watch expressions and similar can be used. The other categories are macros executing either in a Ruby or in a Python interpreter.

Within the repository browser, you can add new folders using the "New folder" button. Before you do so, select the branch of the repository where you want to create the new folder. The new folder will be highlighted and the name can be edited.

To delete files and folders, select the file or folder in the browser and click the "Delete" button (the "x"). Caution: deleting a file or folder cannot be undone currently.

To rename a file or folder, select the entry in the browser and click on the "Rename" button and enter the new name. To move files or folder, simply drag and drop them to the desired target location.

Pain ruby scripts can be imported into the repository as KLayout macro files. Select the branch where you want to import the files in the repository browser, click on the "Import" button and select the file to import. ".rbm" files will automatically be set to "auto-run".

Configuring macros

A macro can be configured in several ways:

- Description: a macro can be given a description text. This text is shown in the repository browser and in the menu if a macro a configured to be shown in the menu.
- Version: the version will be used in a later stage to check dependencies between packages. Currently the version is not used. When used, enter a string of the form "v0.v1.v2" where v0 is the major version, v1 the minor version and v2 the patch. For example: "1.7.1".
- Prolog and Epilog: these are statements which are executed before the script is started and after the script has been run respectively.
- Run on start-up: if this flag is set, the macro is automatically run when KLayout starts. This feature is required for macros defining libraries for example. "Run early on startup" is a special option which makes sure the macro is run at the very beginning of the process. Such macros can be use to supply classes and features for other macros that use the normal "run on start-up" mode.
- Show in menu: with this flag, KLayout will show the macro in the "Macros" menu. That is a simple way to run macros outside the IDE.
- Bind to a key: with key binding, a macro can be run when a certain key combination is pressed. That is another way to run a macro outside the IDE.

The macro properties can be edited using the "Edit properties" button in the editor panel.

Migrating code

Before version 0.22, KLayout did not have a concept of macro IDE, macro folders, interpreters and generic macros. Ruby code written for the pre-0.22 system can be imported into the macro management system of KLayout using the "Import" function from the repository browser toolbar. ".rbm" files are marked as "autorun" in order to emulate the behavior of KLayout 0.21 and previous versions. ".rb" files are converted into KLayout macros without any further assumption. ".rb" files can also simply be copied into the macro directories. However, such files are regarded as secondary sources in KLayout. Typically those are files that are loaded by other macros. Importing a file makes code a generic source (".lym" file) with enhanced capabilities.

After importing the code the macro can be attached to a menu entry by setting "Show in menu" on the properties page. It is also possible to assign keyboard shortcuts. If either the menu item is selected or the key specified in the shortcut is pressed, the macro is executed.

This feature is more convenient to use than the scheme used previously. Until KLayout 0.22 it was only possible to register a macro with a specific menu entry by performing the registration in the code. The disadvantage of this approach is that when the macro code is re-executed, the menu item is registered again and appears in the menu twice. It is therefore more convenient to remove the menu registration code and let KLayout register the macro in the menu by setting the menu binding properties accordingly.

Hints

The editor and IDE can be configured using the "Setup" button in the editor panel. That dialog allows configuring fonts and colors to be used in the editor and the behavior of the debugger in some respects.

The documentation of the Ruby and Python classes exposed by KLayout is available in KLayout's online help system here.

2.12. Macros in Menus

KLayout allows installing macros as menu entries. Each time that menu entry is selected, the macro will be executed. To configure a macro for installation in the menu, open the macro in the macro development environment and open the macro properties dialog.

The macro is configured to be shown in the menu by checking the "Show in menu" check box. That also enables two other fields: the menu path and the group name.

The menu path specifies the position where the macro shall be put. A macro path is a sequence of symbolic names separated by a dot. For example:

- `edit_menu.end` is the end of the "Edit" menu
- `edit_menu.undo` is the "Undo" entry in the "Edit" menu

To obtain a list of the paths available, have a look at the "Key Bindings" page in the "Application" section of the setup dialog ("File/Setup").

The pseudo element "end" denotes the position after the last entry. The same way "begin" indicates the first entry in the menu. The macro will be inserted before the entry indicated by the path. Hence:

- `edit_menu.end`: the macro will be inserted at the end of the "Edit" menu
- `edit_menu.undo`: the macro will be inserted before the "Undo" entry.

If a plus sign follows the macro path element, the new element is inserted **after** this element. For example:

- `edit_menu.undo+`: the macro will be inserted after the "Undo" menu item.

A special form can be used to generate new groups: if the given element does not exist, the menu generator can be instructed to create it by appending the insert point plus the new text string to the element after a ">" character. For example:

- `edit_menu.my_group>end("My Edit Functions").end`: will look for "my_group" and add the new element at the end of this group. If no such group exists, it will be created at the end of the "Edit" menu with the title "My Edit Functions".

If no macro path is specified, the macro is inserted in the "Macros" menu. The description of the macro is used as the menu title. If a shortcut is specified, that shortcut is used for the macro entry as well.

The group name can be used to group together all menu entries with the same group name. Any text is allowed here. A group is separated from the other entries in the menu by a separator line. It is recommended to use the group feature in conjunction with a "end"-terminated menu path which is identical for all entries of the same group. Other uses cases are possible, but the result is not defined.

2.13. About Libraries

Starting with version 0.22, KLayout offers a library concept. Libraries are a way to import cell layout into a layout from the outside and thus a convenient way to provide standard building blocks for the layout. Using a cell from a library is easy: when asked for a cell, select the library where to take the cell from and choose a cell from that library.

Libraries are basically just foreign layouts that are virtually linked to the current layout. When a cell is imported from a library, it is copied into the current layout, so that the current layout by itself is a valid entity.

When a layout containing library references is saved, KLayout stores some meta information in that file which allows it to restore the library links and related information. For GDS, that meta information is stored in a separate top cell. For OASIS, the meta information is stored in special per-cell properties. For other formats, the meta information is not stored currently.

Libraries can be provided in several ways:

- **As ordinary layout files:** Such libraries are simple layout files (GDS, OASIS or other support format). KLayout looks up those libraries in the "libraries" subfolders of the search path and gathers all layout files it finds there into the library repository.

  The search path usually includes the installation site (where the KLayout executable resides) and the application folder (i.e. "~/.klayout" on Linux). Hence libraries can be installed locally (i.e. in "~/.klayout/libraries") or globally (at the installation site).

  For GDS files, the library name will be the LIBNAME of the GDS file. Otherwise it will be the name of the library file minus the extension.

- **Coded libraries:** Such libraries are provided by code, either through shared objects/DLL's or through Ruby code. Basically such code has to provide a layout object containing the library cells. A coded library can also provide PCells (parametrized cells) as library components. Sell About PCells for details about parametrized cells.

Starting with version 0.25, libraries can be provides through packages. This means, they can be downloaded from some repository and can be managed within the package manager. Library installation is very simple this way. Library deinstallation too. See About Packages for details about packages.

2.14. About PCells

Starting with version 0.22, KLayout offers parametrized cells (PCells). PCells are a feature found in other tools to simplify layout by providing generators for common layout building blocks. Parametrized cells do not contain static layout but are created dynamically by code using a given set of parameters.

For example, a circle PCell requires two parameters: the layer where the circle should be produced and the radius of the circle to produce. The code is responsible for creating the circle from these parameters.

Using a PCell is easy: choose the library and the PCell from that library when asked for the cell in the instance options dialog. For PCells, KLayout offers an additional parameters page where it asks for the parameter required by the PCell. The placement of the PCell is done as for simple instances. PCells offer the same instantiation options that normal cells.

KLayout provides a simple library called "Basic" with some useful basic PCells. See About The Basic Library for more details about that library.

Unlike other tools, KLayout offers the unique feature of "guiding shapes". A guiding shape is some kind of "ghost shape" that is not produced as real layout but is present as a part of the PCell instance. It is drawn in the style of the cell frame but can be edited as a normal shape. In particular, a guiding shape can be manipulated with the properties dialog and the partial edit mode. A special case is a point-like shape which can act as a handle of the PCell. In move mode, these shapes can be moved to change the parameter related to that handle.

Another use case for guiding shapes is the rounded path. This PCell uses a path as the input shape and applies rounding to the path's spine corners to compute a new path which smoothly bends around the corners. The radius of the bends is a numerical PCell parameter while the input shape controlling the geometry and the width of the path is the guiding shape.

A PCell implementation consists of at least three parts: a description text, a parameter declaration and a production callback. In addition, a PCell can provide a method that "fixes" parameters according to the PCells consistency rules (coerce parameters). Technically, a PCell is a class implementing a certain interface with these methods.

PCells are usually packed in libraries. PCell libraries can be provided as shared objects/DLL's (in C++) or as Ruby scripts. Because PCell code is only executed if required, performance usually is not the main objective. A Ruby implementation will therefore be sufficient in most cases and is a much easier to maintain. The Ruby approach also benefits from KLayout's integrated development environment.

For an introduction into PCell programming with Ruby, see Coding PCells In Ruby.

2.15. About The Basic Library

The "Basic" library

The "Basic" Library provides some useful generic PCells. One use model is to draw a shape and convert the shape to one of the provided PCells. This use model is suitable for creating Circles, Ellipses, Donuts, Texts and rounded and stroked polygons or rounded paths.

To use that feature, draw the shape and choose "Convert To PCells" from the "Edit"/"Selection" menu. A dialog will be shown where the target PCell can be selected. Only those PCells supporting that shape type will be shown.

The "Basic" library provides the following PCells:

- **TEXT**: A text generator
- **CIRCLE**: A circle
- **DONUT**: A donut (circle with hole)
- **ELLIPSE**: An ellipse
- **PIE**: A pie (a segment of a circle)
- **ARC**: An arc (a segment of a donut)
- **ROUND_PATH**: A rounded path (a path bending around the corners with a given radius)
- **ROUND_POLYGON**: A rounded polygon (a polygon with rounded corners)
- **STROKED_BOX**: A stroked box (the "rim" of a box, optionally with smooth corners)
- **STROKED_POLYGON**: A stroked polygon (the "rim" of a polygon, optionally with smooth corners)

TEXT

The text generator can produce texts in various forms. The following sample shows inverse text, normal text and text with bias and enlarged character spacing:

It's even possible to install custom fonts. Fonts are basically GDS files with the following features:

- One cell per character. Cells must be either named like the character "A", "B", "0" etc. or like the ASCII code in 3-digit decimal notation (i.e. "036" for the dollar character).
- The characters must be drawn in the character cells on layer 1/0. A box defining the extension of the characters (including spacing) must be drawn on layer 2/0. Optionally a background rectangle for the "inverse font" feature can be drawn on layer 3/0.
- One cell called "COMMENT" with text objects defining the basic text properties through their strings, in particular:
- **design_grid=x**: specifies the basic grid the characters are designed on. "x" is the grid in database units.
- **line_width=x**: specifies the line width in database units.
- A comment string which is displayed in the font selection box on the PCell parameters page.

Custom fonts are installed by copying the font file to a folder named "fonts" in one of the places in KLayout's path. The standard font can be found in "src/db/db/std_font.gds" in the source package.

CIRCLE and ELLIPSE

These PCells define a circle and an ellipse. In both cases, the number of interpolation points (per full circle) can be specified. The default is 64 points. A circle features a handle to define the diameter. An ellipse features two handles defining the diameters in x and y direction.

When a shape is converted to a circle or ellipse PCell, the shape's bounding box will be used to define the enclosing box of the circle or ellipse.

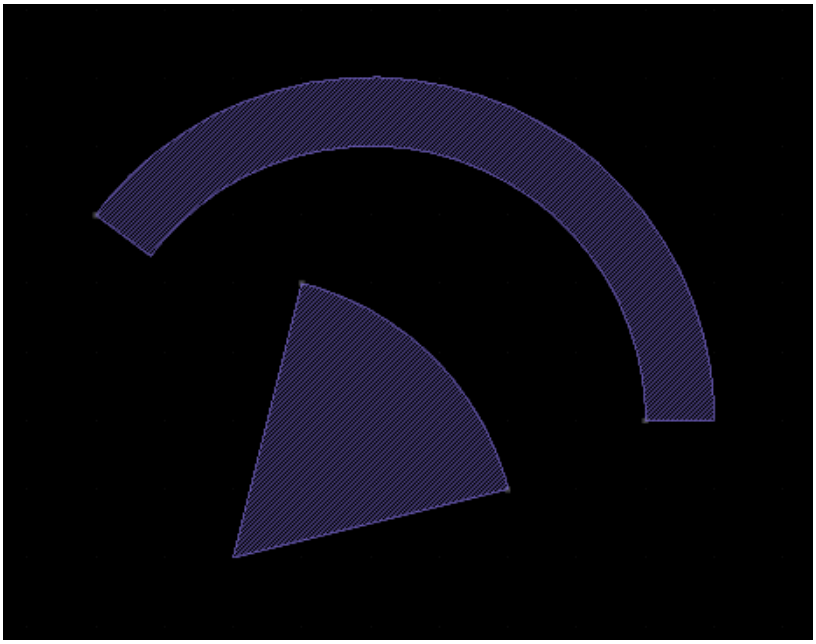
DONUT

The donut PCell creates a circle with a hole. This PCell features the same parameters than the circle but an additional parameter defining the hole radius. For that, it provides two handles - one for the outer and one for the inner radius.

When a shape is converted to a donut, the shape's bounding box will be used to define the enclosing box of the donut and the hole's diameter will be chosen to be half of the outer diameter.

PIE and ARC

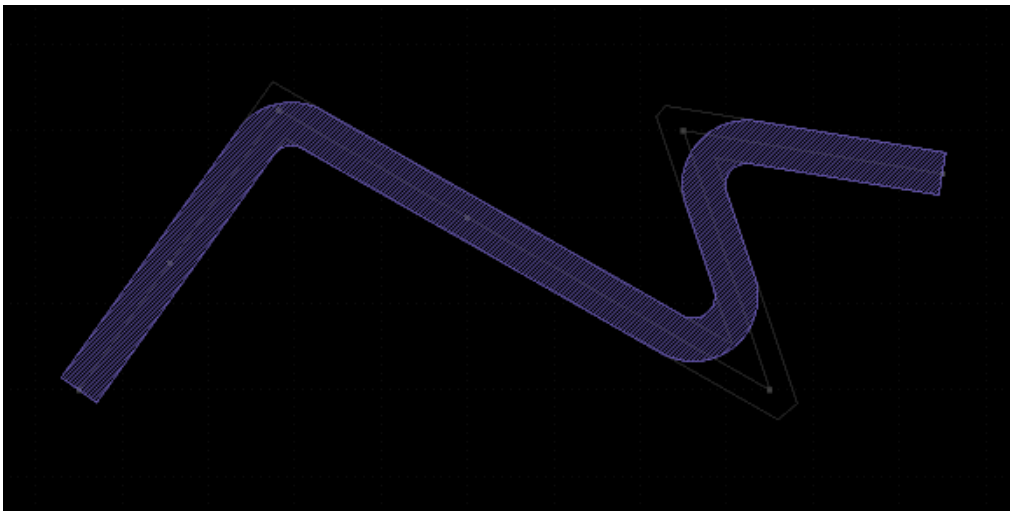
Both of these PCells are segments of circles or donuts. The PIE PCell features two handles to define the radius and start and end angle. The ARC PCell also features two handles to define outer and inner radius as well. The following image shows PIE and ARC in action:



Both PCells do not support conversion of shapes.

ROUND_PATH

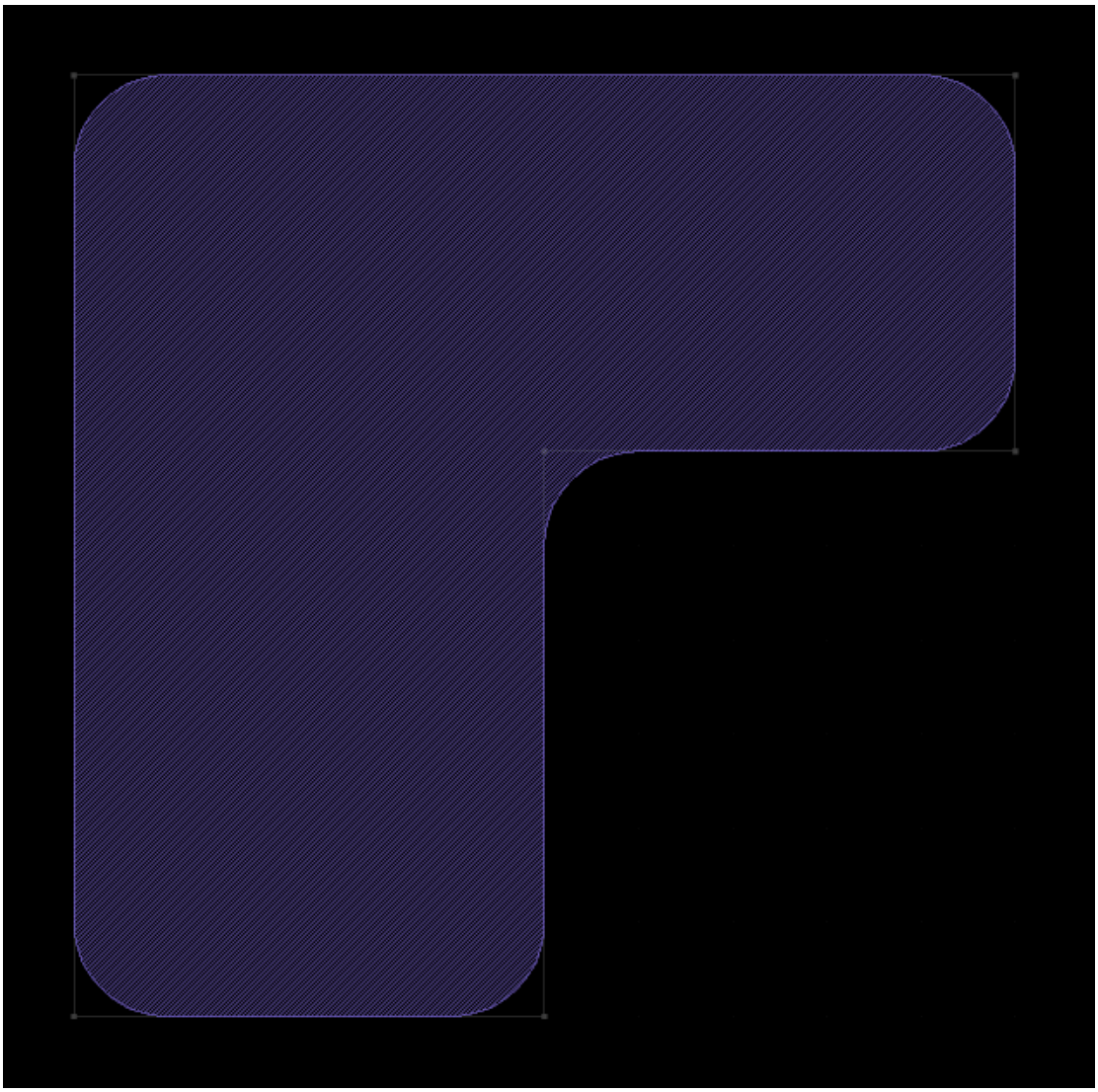
The round path is a PCell that is based on a path object but is capable of smoothing the path's corners by applying a radius. The following image gives an example:



The PCell features a parameter that defines the radius. The path itself can be manipulated as usual, in particular with partial edit mode. Path objects can be converted to ROUND_PATH pcells. In that case, the initial radius will be chosen to be roughly 10 percent of the minimum bounding box dimensions of the original path.

ROUND_POLYGON

The round polygon is a PCell that is based on a polygon object but is capable of smoothing the polygon's corners by applying a radius. The following image gives an example:



The PCell features a parameter that defines the radius. The polygon itself can be manipulated as usual, in particular with partial edit mode. Polygon, box or path objects can be converted to ROUND_POLYGON pcells. In that case, the initial radius will be chosen to be roughly 10 percent of the minimum bounding box dimensions of the original polygon.

STROKED_POLYGON or STROKED_BOX

The stroked polygon or box is a PCell that is based on a polygon object but will produce the "rim" of this polygon. In addition, it can apply corner rounding with a given radius.

The PCell features two parameters that define the radius and width of the "rim" line. The polygon or box itself can be manipulated as usual. Polygon, box or path objects can be converted to STROKED_POLYGON or STROKED_BOX pcells. In that case, the initial radius will be zero. The width of the rim line will be chosen to be roughly 10 percent of the minimum bounding box dimensions of the original polygon. For STROKED_BOX, the bounding box of the original shape will be used as the basic shape.

2.16. About Packages

"Salt" is KLayout's package manager which allows selecting and installing packages from a global repository. Packages make KLayout more tasty. Packages (the "grains") may cover a variety of features:

- Ruby or Python macros
- DRC runsets
- Technologies
- Fonts for the Basic.TEXT PCell
- Static layout libraries
- PCell libraries
- Code libraries for Ruby and Python
- Binary extensions

Packages can depend on other packages - these are installed automatically if a package requires them and they are not installed yet.

Packages are identified by name. A package name needs to be unique in the package universe. You can use a prefixed name like `www.mydomain.org/nameofpackage` to create a non-ambiguous name. Use a slash to separate the prefix from the actual package name. The choice of the prefix is entirely up to you as long as it contains letters, digits, underscores, hypthens or dots. You can use a domain name that is owned by yourself for example. You can use multiple prefixes to further differentiate the packages inside your namespace.

Packages also come with version information, so KLayout can check for updates and install them if required. KLayout will assume strict upward compatibility. This specifically applies to packages that other packages are depending on (such as code libraries). If you need to change them in a non-backward compatible way, you'd need to provide a new package with a different name.

Packages come with some meta data such as authoring information, an optional icon and screen shot image, license information and more. The more information you provide, the more useful a package will become.

The key component for public package deployment is the "Salt.Mine" package repository service. This is a web service that maintains a package index. It does not host the packages, but stores links to the actual hosting site. In order to author a package, you need to upload the package to one of the supported host sites and register your package on the Salt.Mine page. Registration is a simple process and the only information required is the link to your host site and a mail account for confirmation.

Installing Packages

To install external packages, open the package manager with "Tools/Manage Packages". On the "Install New Packages" page, a list of available packages is shown. Select the desired packages and mark them using the check mark button. Marked packages will be downloaded and installed with the "Apply" button.

A filter above the package list allows selecting packages by name. The right panel shows details about the package currently selected.

Updating Packages

To check for updates, use the "Update Packages" tab of the package manager. In the list, those packages for which updates are available are shown. Mark packages for update using the check mark button. Click "Apply" to apply the selected updates.

Uninstalling Packages

To uninstall packages, open the package manager using "Tools/Manage Packages". Go to the "Current Packages" tab. Select a package and use the "Remove Package" button to uninstall the package.

Creating Packages

For package development you can utilize KLayout to initialize and edit the files inside the package folder or populate the folder manually.

KLayout offers initialization of new packages from templates. You can modify that package according to your requirements afterwards. To create a package from a template, open the package manager using "Tools/Manage Packages", go to the "Current Packages" tab and push the "Create (Edit) Package" button. Chose a template from the list that opens and enter a package name (with prefix, if desired). Select "Ok" to let KLayout create a new package based on the template you selected.

The package details can be edited with the "pen" button at the top right of the right details panel. Please specify at least some author information, a license model and a version. If the package needs other packages, the dependencies can be listed in the "Depends on" table. Those packages will be automatically installed together with the new package. The showcase image can be a screen shot that gives some idea what the package will do. The package details are kept in a file called "grain.xml" inside the package folder. You can also edit this file manually. The "grain.xml" is the basic description file for the package.

If the package is a macro or static library package, the macro editor can be used to edit the package files. If the package is a tech package, the technology manager can be used to edit the technology inside the package. To populate the package folder with other files use your favorite editor of KLayout itself for layout files.

Deployment inside your organisation

Once a package is finished, it needs to be deployed to make it available to other users. Deployment basically means to put it on some public place where others can download the package. For local deployment inside an organisation, this can be a web server or a folder on a file server. KLayout talks WebDAV, so the web server needs to offer WebDAV access. A subversion (SVN) server provides WebDAV by default, so this is a good choice. For the packages themselves, Git or WebDAV/Subversion can be used. You need to specify the protocol in the package URL (see below).

After a package has been made available for download, it needs to be entered in the package index. For local deployment, the index can be a file hosted on a web server or on the file system. The package index location needs to be specified by the `KLAYOUT_SALT_MINE` environment variable which contains the download URL of the package index file.

For public deployment, the Salt.Mine service (`http://sami.klayout.org`) is used to register new packages in the package index. By default, KLayout loads the package index from that service, so once your package is registered there, everyone using KLayout will see it.

The Package Index

Public Packages are published on the Salt.Mine server. This is a web service that delivers a packages index with some meta data such as current version, the icon and a brief description. KLayout uses this list to inform users of packages available for installation and available updates. For local deployment, the package index can be served by other ways too. The only requirement is to be accessible by a http, https or file URL.

The basic format of the index is XML with this structure:

```
<salt-mine>
  <salt-grain>
    <name>name</name>
    <version>Version</version>
    <title>Title of the package</title>
    <doc>A brief description</doc>
    <doc-url>Documentation URL</doc-url>
    <url>Download URL</url>
    <license>License model</license>
    <icon>Icon image: base64-encoded, 64x64 max, PNG preferred</icon>
  </salt-grain>
  ...
  <include>URL to include other index files into this one</include>
  ...
</salt-mine>
```

You can include other repositories - specifically the default one - into a custom XML file. This allows extending the public index with local packages.

When the package manager is opened, KLayout will download the index from `http://sami.klayout.org/repository.xml`. You can set the `KLAYOUT_SALT_MINE` environment variable to a different URL which makes KLayout use another dictionary service, i.e. one inside your own organisation. This service can be any HTTP server that delivers a package list in the same format than the Salt.Mine package service. The URL can also be a "file:" scheme URL. In this case, KLayout will download the list from the given file location.

When installing a package, KLayout will simply download the files from the URL given in the package list. KLayout employs the WebDAV protocol to download the files. This protocol is spoken by Subversion and GitHub with the subversion bridge. The latter requires a simple translation of the original Git URL's to obtain the subversion equivalent.

Generic package URLs

Package URLs are the locations from where KLayout will download the package files. The package data is a hierarchy of files. The package description file ("grain.xml") needs to be located at the root of this hierarchy.

A plain package URL will instruct KLayout to download the package file from the given hierarchy using the WebDAV protocol. You can add "svn+" at the front of the URL to make this specification a little more explicit. These are WebDAV/Subversion URLs:

```
https://www.klayout.org/example-package
svn+https://www.klayout.org/example-package
```

Git is another option to host packages. In order to use the Git protocol, add "git+" at the front of the URL:

```
git+https://github.com/klayout/example-package.git
```

This will make KLayout clone the "example-package" repository and use the default branch for checking out the files. It will expect a file named "grain.xml" at the root of the package file hierarchy.

To use a different branch or tag, specify the tag or branch name in square brackes. For example:

```
git+https://github.com/klayout/example-package.git[v1.0]
```

This will use the "v1.0" tag or branch for checking out the files.

It is also possible to have packages in the a subfolder of the repository. To specify the subfolder, separate the subfolder path from the repository URL using "+", like this:

```
git+https://github.com/klayout/repository.git+klayout/package-data[v1.0]
```

This will clone the "repository" project and in this project, KLayout will checkout files from "klayout/package-data" and below. This is also the place, where KLayout expects the "grain.xml" file.

Installation Hooks

Scripts can register an event through Application#on_salt_changed which indicates that packages have been installed or uninstalled.

Packages itself can supply special scripts which are executed after a package was installed or before a package is uninstalled:

- `_install.lym`: if present, this script is executed after the package is installed.
- `_uninstall.lym`: if present, this script is executed before the package is uninstalled.

Both scripts need to be stored in the same location as "grain.xml" and have to use "lym" format. This is the generic XML script format KLayout employs as an interpreter-agnostic script representation.

2.17. About Technology Management

Technology management summarizes features which require a certain interpretation of a layout. In particular, layout layers are assigned a physical meaning, for example via layers or active area layers in CMOS technologies. Since that interpretation often is depending on the technology the product will be fabricated with, the ability to provide multiple setups is summarized as "technology management".

A technology setup implements the following aspects:

- Layer mapping: when the layout reader loads a file that for a certain technology, it can apply a layer mapping, i.e. apply layer names to GDS layers, filter layer etc.
- Layer properties: depending on the technology, the layer display can be configured by providing a technology specific layer properties file.
- Connectivity: the layer stack and the connections made by layers for the net tracer feature.
- Macros: macros associated with the technology. When the corresponding technology is selected (is the one of the current layout), such macros will show up in the menu if they are bound to a menu entry. Otherwise they will be invisible.
- DRC scripts: in the same way, DRC scripts can be associated with a technology.
- Libraries: if a library is associated with a technology, it is shown in the list of available libraries when an instance is created. Library association cannot be edited. Instead, a library installed in the system comes with a technology association itself.
- File format options: technology specific file reader or writer options can be given. When a layout is saved, it will use the writer options from its technology. When loading a layout, the reader options from the active technology will be used.

In the future, more aspects may be added to the technology definition.

There is always one "Default" technology that is used when no technology is specified.

Setting up technologies

Technologies can be set up using the "Technology Manager" in the "Tools" menu. There is always a "Default" technology which provides the settings when no technology is selected. New technologies can be added or technologies can be deleted using the "+" or "x" buttons below the technology tree.

A technology has a name (a short string) and a description. The name is used to identify the technology in various places. The description is the human-readable text that is displayed in the technology selection boxes for example. The short name can be changed by selecting the technology and pressing the "Rename" button or using "Rename" from the technology tree's context menu (right mouse click). The description can be edited on the "General" page.

In the technology manager, below each technology, the components are shown that define the various aspects of a technology. Beside the "General" aspect (names, descriptions) there is a "Layers" component which defines the layer mapping table and layer properties file and the "Connectivity" component which defines the settings for the net tracer.

Using technologies

When more than the default technology is defined, KLayout provides a drop-down menu in the tool bar to select the current technology. The current technology is the technology used when new files are loaded. It is also possible to define the technology to be used on the command line using the "-n" switch (applies to following files and specifies the technology to use by their short name).

The technology of the currently selected layout is shown in the status bar of the main window in the left section. It is possible to switch the technology of a layout already loaded by using the "Layout Properties" dialog from the "File" menu. After switching the technology, the layer properties defined in the technology can be applied and the associated macros or DRC scripts are shown in the menu if they are associated bound to a menu entry or the key binding becomes active if a shortcut is defined for that macro.

Technologies and macros or DRC scripts

Macros or DRC scripts are stored in sub-folders relative to the technology's base path. When no base path is specified or the base path is invalid, macros or DRC scripts cannot be associated. KLayout will look search for macros, if a directory called "macros" is present in the base path. If it finds files with a valid macro suffix there it will associate them with this technology. The same way, KLayout will look for DRC scripts, if a directory called "drc" is present in the base path.

Macros and DRC scripts associated with a technology are shown in the technology manager. To edit or debug scripts of macros, use the macro development environment ("Macros/Macro Development"). If a technology has a macros or DRC scripts folder, the macro or DRC scripts tree in the development environment will show a corresponding top-level branch for that technology.

Multiple technologies can share the same base path - hence it is possible to share macros or DRC scripts between technologies.

Managing technologies

Technologies can be imported and exported to technology files (suffix ".lyt"). This is mainly useful to exchange technology settings between users or technologies.

Except for the default technology, technologies are kept in technology folders in KLayout's application path. They are read from subfolders from the "tech" directories. The technology definition itself is held in a file with extension ".lyt". The technology folder may have subfolders to hold library files, macros, DRC runsets, LEF files and other pieces of the technology package.

Technologies can be managed using packages. Packages are a convenient way to share add-ons between users. Packages can be installed from a common repository and allow easy addition and removal of components. Technologies are one aspect of packages, so it's possible to create packages that contribute one or more technologies. See About Packages for more details.

2.18. About Custom Layout Queries

Layout queries are an advanced feature of KLayout which provides a very generic method to manipulate or search the geometrical or cell information of a layout. The basic concept of custom queries is borrowed from SQL, the language widely used for accessing databases. Instead of working on linear tables with rows and columns, KLayout's queries work on the layout structure which basically a cell tree, a layer set orthogonal to that and per-cell/per-layer geometrical information which itself is divided into shape classes.

Layout queries have a layered structure, like an onion: the core of the query is a cell query which selects one or many cells with or without their children. The cell query can be wrapped by a shape query which addresses the shapes of the selected cells or instances, optionally confined to specific layers. The last layer is formed by the action: this is the activity that KLayout will perform on the selected objects. The default action is simply to report the results. It is also possible to delete the selected objects or to perform a custom operation on them.

On cell and shape level, conditions can be specified which will restrict the operation on a subset of the selected objects. Reports can be sorted by a arbitrary key derived from the current object.

On cell level three different relationship models are supported:

- Individual cells: no hierarchy is involved
- The cell tree: the cell tree is the parent/child relationship without explicit instantiation information
- The cell instance tree: every individual instance of a cell is considered

Expressions play an important part in layout queries, both as actions (assignment type expressions) as well as conditions or for the derivation of sorting keys. See About Expressions for details about expressions. Within expressions, RBA objects are used to represent shapes or instances (see Class Index for a list of classes available). Depending on the context of the query, a variety of functions is available to access the properties of the current item and context.

The key to layout queries is the "Search and Replace" feature: this dialog uses custom queries to emulate simple search and replace functionality on the first three tabs of the dialog ("Delete" is regarded as a special kind of "replacement" here). However, the true power is revealed on the forth page: here you can enter all kind of custom queries. Clicking on "Execute" will run the query and display the results in the right panel.

If a search or replace action is specified on the first three tabs, the corresponding custom layout query will be shown in the entry box of the forth one. That way it is very easy to create a first query using the standard functions, switch to the custom query page and adjust it to fit the specific requirements.

Building queries: cells

The very core of a query is a cell expression. The most simple form of a cell expression is a simple cell name. This expression will select the cell called "RINGO":

```
RINGO
```

Cell expressions can contain wildcard in the "glob" form made popular by the Unix and Windows command line. "*" is for an arbitrary sequence of zero to many characters, "?" for any single character. "{A,B,C}" is for either the character sequence "A", "B" or "C", "[ABC]" is any of the characters "A", "B" or "C" and "[^ABC]" is for any character not "A", "B" or "C". Round brackets can be used to group parts of the string for later reference. If brackets of any kind are used inside a match string, either single or double quotes should be used around match strings in order to avoid ambiguities with other parts of the query syntax.

This expression will select all cells starting with "T":

```
T*
```

Although it is not necessary to do so, it is recommended to mark a cell query explicitly as such using the optional "cells" or "cell" keyword. This query has the same effect than the previous one but is somewhat more robust if used in nested queries we will learn about later:

```
cells T*
```

A cell expression can already be used by its own. The report of such a query will simply contain the cells selected by that expression. If combined with an action, such expressions already provide useful manipulation functionality.

The "delete ..." action will delete the given cells:

```
delete cells T*
```

The "with .. do ..." action can be used to manipulate the cell. This example will rename the cell by replacing the "T" prefix with a "S". The part after "do" is an expression which is evaluated for each hit. Note that "$1" is used to refer to the first matching bracket in the last match. "cell.name" is a method call on the object "cell" which is provided by the query in the context of a cell query. "cell" is a "Cell" object (see Cell) and setting the name will basically rename the cell. The expression used for the assignment will put an "S" in front of the rear part of the name, hence replace "T" by "S":

```
with cells "T(*)" do cell.name = "S"+$1
```

Note the quotes around the "T(*)" match expression. They are necessary to make the brackets part of the match expression rather than the cell query. It is usually a good idea to put the match expression inside quotes to avoid ambiguities.

The last example already demonstrates how a combination of two simple concepts - simple cell queries and expressions - form a new and very generic feature. We will soon learn about the power of that concept.

Building queries: cell trees

Cell queries are the most simple form of core queries. The next level is entered by extending the concept to hierarchies. Cell hierarchies come on two flavors: a parent-to-child relationship tree (the cell tree) and the instantiation tree. In the cell tree, each cell is at most present once in the context of a parent cell, independent of the number of times a cell is used inside a parent cell. The cell tree just describes the fact that a cell is a child cell of another, not how the cell is used. The instance tree adds this detailed information as well: how many times a cell is used and what transformations are applied per such instance.

The cell tree can be accessed within cell queries using the "." operator to separate parent and child cell. The following cell query returns all cells which are children of a cell "A":

```
cells A.*
```

Multiple levels may be nested, for example the following query lists all cells which are second-level children of the "A" cell:

```
cells A.*.*
```

Such expressions form a "path" leading from an initial cell to some cell, which is returned by the query. The "." separates the path elements like the slash or backslash does in a file path.

Top cells can be addressed by a leading "." similar to the leading slash of an absolute file path in Unix. The following query will return all top cells:

```
cells .*
```

Brackets can be used to group parts of the path. That has no immediate effect, but it can be usedful in combination with quantifiers and branches as we will see soon. The following queries are equivalent:

```
cells TOP.*.A
cells TOP(.*.A)
cells TOP(.*)(.A)
```

Please note that brackets can only be put between the dot and the previous element. A query like "cells TOP.(*.A)" is invalid since the opening bracket is after the dot.

Alternative paths can be specified by separating them with a comma. Such alternatives must be put inside brackets. For example, this query selects all cells that are children of "TOP" and start with an "A" or which are second-level children and start with an "E":

```
cells TOP(.A*,.*.E*)
```

Path elements can be made optional with a "?" symbol and expanded an variable number of times using quantifiers like "*" (0 to many) and "+" (one to many) or "{n,m}" (n to m times). Note that you'll have to put the expression subject to the quantifier in brackets in order to avoid ambiguities of the star operator. The following expression will return the A cell in every possible child context of "TOP", i.e. as direct child, second-level child and so on:

```
cells TOP(.*)*.A
```

To understand that query, note that the "*" inside the brackets is forming the match string while the outer star is forming the quantifier. That query reads in expanded form "TOP.A", "TOP.*.A", "TOP.*.*.A" etc.

There is a useful abbreviation for the above case. The following query will also produce "A" in every child context of "TOP":

```
cells TOP..A
```

The double dot operator matches an arbitrary part of the instantiation path before and after a cell even without being anchored at one end. Used before a cell name, it will return all contexts a cell is used in (including top cells and all child contexts). Used after a cell, it will return the cell plus all child cells in each possible context. Used before a cell it will deliver all contexts that cell is used in every top cells. The following query will deliver "TOP" plus all its direct and indirect children:

```
cells TOP..
```

Note that the previous query may deliver the same cell multiple times - once for each context (call path from TOP) it is used in. Hence "TOP.." will basically expand into the cell tree with "TOP" as the root.

In order to get the names of all cells called from a given cell, you can use the "select" action with the cell name and the "sorted by .. unique" output selector to remove duplicates of cell names:

```
select cell_name of cells TOP.. sorted by cell_name unique
```

See below for a description of the "select" action.

Within a path, dynamically computed components can be inserted using the "$(..)" notation which wraps an expression. That expression is evaluated in the context of the previous path component. For example, the following query selects all child cells which are named like their parent with an "A" prefix:

```
cells *.$("A"+cell_name)
```

Building queries: instances

Cell trees can be expanded into instance trees simply by prepending "instances". This will deliver all direct instances of "TOP":

```
instances of TOP.*
```

When asking for instances, more information is available inside the query. For example, the instance's orientation and position is available. With the "instances" specification, array references are expanded into single instances. To keep arrays as such, use "arrays" instead of "instances":

```
arrays of TOP.*
```

Cell or instance queries can be filtered using the "where" clause. After the "where" an expression is expected with a number of predefined variables that reflect the context (see below for the variables available). The following query selects all child cells of "A" where the cell name has a length of 5 characters:

```
cells A.* where len(cell_name)==5
```

Building queries: shapes

So far we have dealt with cells and their instantiations. We enter the next level now by introducing shapes.

Shape queries are built atop of the cell/instance level. A simple example selects all shapes of the cell "TOP":

```
shapes of cell TOP
```

Shape queries can be confined to certain shape types. For example, this confines the query to boxes:

```
boxes from cell TOP
```

Allowed shape type are "boxes", "polygons", "texts" and "paths". In the context of a shape query additional variables are available for expressions. The most important one is "shape" which is a Shape object (see Shape) That objects provides access to the shape addressed in a generic way. Specialization to a specific shape type is possible through the shape specific accessor methods (i.e. "shape.box_width") or the specific objects (i.e. "shape.box").

Multiple shape types can be given with "or" or a comma:

```
boxes or polygons from cell TOP
```

Shape queries can be confined to certain layers. This query will report all shapes from layer 8, datatype 0:

```
shapes on layer 8/0 from cell TOP
```

Intervals can be specified with the hyphen ("-") and multiple layers or intervals can be listed with a comma or semicolon. The following will list the shapes from layer 8, datatype 0 to 10 and layer 9, datatype 0 only (note that "no datatype" is interpreted as datatype 0):

```
shapes on layer 8/0-10, 9 from cell TOP
```

For formats that support named layers only (i.e. DXF), the layer name can be given. The following query lists shapes from layers METAL and POLY (case sensitive!):

```
shapes on layer METAL, POLY from cell TOP
```

Any kind of cell query can be used inside the shape query. If a cell query renders multiple cells, the shape query will be applied to each of the cells returned. If instances are selected by the cell query, the shapes will be reported for each instance. Since the cumulated transformation of a specific instance into the top cell is available through the "path_trans" (database units) or "path_dtrans" (micrometer units) variable, it is possible to transform each shape into the top cell in the instance case. The following expression combines a "with .. do" action with a shape query to flatten all shapes below "TOP":

```
with shapes on layer 6 from instances of TOP.. do
 initial_cell.shapes(<10/0>).insert(shape).transform(path_trans)
```

That expression reads all shapes of cell "TOP" and its children, inserts them into a new layer 10, datatype 0 and transforms the shape after it has been inserted. This expression makes use of the variables "initial_cell" (a Cell object representing the root cell of the cell query), "shape" (a pointer to the currently selected shape and "path_trans" (a Trans object representing the transformation of the current shape into the root cell of the query). It also employs the angle bracket layer constant notation which specifies a layer in the target notation and can be used in place of the layer index value usually used inside the API. Note that the target layer must exist already, i.e. must have been created in "Edit/Layer/New Layer" for example.

Shape queries can be confined with conditions. A condition is entered with a "where" clause plus an expression that selects the shapes. This condition selects shapes with an area of more than 4 square micron (note that the "um2" unit must be given, since it will cause the value to be converted into the database units used internally):

```
shapes from cell TOP where shape.area < 4 um2
```

Shape conditions can be combined with cell conditions. To avoid ambiguities, the cell query must be put into brackets in that case:

```
shapes from (cells * where len(cell_name)==4) where shape.area < 4 um2
```

Actions

Actions specify operations that are to be performed on the results of a query. The default action is to just list the results. In the "Search and replace" dialog, the results will be listed right to the query entry box as a table. Depending on the context of the query, cell names, cell names plus parent cell, cell instances or shapes are listed.

"select" action

The "select" action will compute one or more results from each item returned by the query and present the computed value in a table. The general form is:

`select` expr1, expr2, ... **`from`** query

"expr1", "expr2" ... are expressions. For example this action computes area and perimeter for all shapes of cell "TOP":

```
select shape.area, shape.perimeter from shapes of cell TOP
```

The "select" action offers sorting with optional reduction to unique values:

`select` expr1, expr2, ... **`from`** query **`sorted by`** sort_key

`select` expr1, expr2, ... **`from`** query **`sorted by`** sort_key **`unique`**

Here "sort_key" is an expression which delivers the value by which the output will be sorted. If "unique" is specified, items with identical sort key are reduced to a single output.

"with" action

The "with" action executes an expression on each item returned by the expression. In that sense it is basically equivalent to the "select" action but the results of the operation are discarded and the intention of the expression is to modify the results. The general form of that action is this:

with query **do** expr

For example, this action will move all shapes of cell "TOP" from layer 6 to layer 10, datatype 0:

```
with shapes on layer 6 of cell TOP do shape.layer = <10/0>
```

"delete" action

This action will simply delete the objects selected by the query:

delete query

For example, this query deletes all shapes from layer 6, datatype 0 on cell TOP:

```
delete shapes on layer 6 of cell TOP
```

Variables available per context

Common variables

The following variables are available in all queries:

Name	Value type	Description
layout	Layout	The layout object that this query runs on.

Cell query context

In the plain cell and cell tree context, the following variables are available:

Name	Value type	Description
path	Array	Array with the indexes of the cells in that path. For a plain cell, this array will have length 1 and contain the index of the selected cell only.
path_names	Array	Array with the names of the cells in that path. For a plain cell, this array will have length 1 and contain the name of the selected cell only.
initial_cell	Cell	Object representing the initial cell (first of path)
initial_cell_index	Integer	Index of initial cell (first of path)
initial_cell_name	String	Name of initial cell (first of path)
cell	Cell	Object representing the current cell (last of path)
cell_index	Integer	Index of current cell (last of path)
cell_name	String	Name of current cell (last of path)
hier_levels	Integer	Number of hierarchy levels in path (length of path - 1)
references	Integer	The number of instances of this cell in the parent cell. Array references count as 1. For plain cells, this value is 0.
weight	Integer	The number of instances of this cell in the parent cell. Array references count as multiple instances. For plain cells, this value is 0.
tot_weight	Integer	The number of instances of this cell in the initial cell along the given path. Array references count as multiple instances. for plain cells, this value is 0.

Name	Value type	Description
instances	Integer	Equivalent to "weight", but also available for plain cells. For plain cells, the value represents the number of times, the cell is used in all top cells.
bbox	Box	The cell's bounding box.
dbbox	DBox	The cell's bounding box in micrometer units.
cell_bbox	Box	Same as "bbox" (disambiguator from shape and instance bounding boxes).
cell_dbbox	DBox	Same as "dbbox" (disambiguator from shape and instance bounding boxes).

Instance query context

In an instance query context, the properties of the current instance are available as variables in addition to most of the ones provided by the cell query context. These variables are not available in instance context: "weight", "references" and "tot_weight". Apart from that these additional variable are provided:

Name	Value type	Description
path_trans	ICplxTrans	The transformation of that instance into the top cell. For a plain cell that is a unit transformation.
path_dtrans	DCplxTrans	The transformation of that instance into the top cell in micrometer units. For a plain cell that is a unit transformation.
trans	ICplxTrans	The transformation of that instance (first instance if an array).
dtrans	DCplxTrans	The transformation of that instance (first instance if an array) in micrometer units.
inst_bbox	Box	The instance bounding box in the initial cell.
inst_dbbox	DBox	The instance bounding box in the initial cell in micrometer units.
inst	Instance	The instance object of the current instance.
array_a	Vector	The a vector for an array instance or nil if the instance is not an array.
array_da	DVector	The a vector for an array instance in micrometer units or nil if the instance is not an array.
array_na	Integer	The a axis array dimension or nil if the instance is not an array.
array_b	Vector	The b vector for an array instance or nil if the instance is not an array.
array_db	DVector	The b vector for an array instance in micrometer units or nil if the instance is not an array.
array_nb	Integer	The b axis array dimension or nil if the instance is not an array.
array_ia	Integer	The a index when an array is iterated (0 to array_na). Not available with instance queries with "arrays of ...".
array_ib	Integer	he b index when an array is iterated (0 to array_nb). Not available with instance queries with "arrays of ...".

Shape query context

In the context of the shape query, the following variables are available in addition to the variables made available by the inner cell query. The inner cell query is either a instance query or a cell query:

Name	Value type	Description
bbox	DBox	The shape's bounding box
dbbox	DBox	The shape's bounding box in micrometer units
shape_bbox	Box	Same as "bbox" (disambiguator for cell or instance bounding boxes)
shape_dbbox	DBox	Same as "dbbox" (disambiguator for cell or instance bounding boxes)
shape	Shape	The shape object
layer_info	LayerInfo	The layer description of the current shape's layer

Name	Value type	Description
layer_index	Integer	The layer index of the current shape

2.19. Notation used in Ruby API documentation

Introduction

The documentation of the Ruby API is derived from the C++ declaration of the specific methods. Hence the notation deviates somewhat from the usual documentation of Ruby methods. In particular the following differences are noteworthy:

- **"Static" methods are "class methods":**

  The C++ term "static" refers to methods available within a class without requiring an object. In Ruby, the term "class method" is commonly used to refer to such methods.

- **Different flavors of object arguments:**

  In C++ there are references, pointers and objects passed by value. In Ruby there are only references. RBA maps the C++ concepts to ruby by allowing the "nil" value only for pointer arguments. Pointer arguments are specially marked in the documentation.

- **There is "constness":**

  C++ has the concept of "const" methods and arguments. In C++, an object reference can be "const", which means that the object behind such a reference cannot be modified. A "const" method is a method which can be called on "const" object reference and such a method may not alter the state of the object.

  A method can have "const" reference arguments which means that objects passed through such arguments are not modified by the method. Such arguments are also said to have "in" semantics. Hence, "const" references can be passed to such arguments. In Ruby there is no concept of "constness" or "in" parameters. Every method is allowed to alter the state of the object it is called on and pf the objects it gets as arguments. In that sense, C++ allows specification of a more constraint "contract" between caller and method that is called. RBA emulates constness in Ruby to some extent and it may disallow calling non-const methods on const references or passing const reference to non-const arguments. Return value can also be const pointers which means that the object returned cannot be modified.

- **Strong typing:**

  In C++, arguments and return values are strongly typed. RBA will check the arguments passed to a method and convert them properly. Hence the type of argument is important. "int" type arguments may not be passed strings for example.

  The type system if C++ is also somewhat more restricted: the value range of an integer argument is limited and for example there are unsigned types which cannot be passed negative values. Hence the type of an argument is noted in the documentation. A particular return type is "void" which basically means "no value returned". Strong typing extends to object references and RBA checks if an argument can be converted to the object required.

- **Virtual methods:**

  In C++, a method must be virtual before it can be reimplemented by a derived class. In Ruby all methods are virtual. Since reimplementing a non-virtual method does not have any effect in RBA, virtual methods are marked as such.

General layout of the documentatation

The documentation states the following methods (in that order):

- Public constructors
- Public methods
- Public static methods and constants
- Protected methods (static, non-static and constructors)
- Deprecated methods (protected, public, static, non-static and constructors)

Deprecated methods are listed for reference only. Use of such methods or constants is not recommended because they might be removed in the future.

Examples

Here are some examples for method documentations (signatures):

- ***[virtual]* bool event(QEvent ptr ev)**:

  A virtual method called "event" returning a boolean value (Ruby "true" or "false") and expecting one argument (a pointer to a QEvent object). "ptr" indicates that the argument is a pointer, "arg1" is the argument name.

- **void add_reference(const RdbReference rdb_ref)**:

  A method without return value which expects one parameter. The parameter must be a reference to the RdbReference object. The reference must not be nil since it is not a pointer, but can be a reference to a const object. The name of the argument is "rdb_ref".

- ***[const]* unsigned int num_items**:

  A parameterless const method called "num_items" that delivers the number of items as an unsigned integer value.

- ***[iter]* RdbReference each_reference**:

  An iterator called "each_reference" delivering RdbReference objects.

- ***[signal]* void layoutAboutToBeChanged**:

  A parameterless signal (event) called "layoutAboutToBeChanged" (see Events And Callbacks for details about events or signals).

- ***[signal]* void objectNameChanged(string objectName)**:

  A signal (event) called "objectNameChanged" with one string argument.

2.20. DRC Reference

2.20.1. DRC Reference: DRC expressions

DRC expression objects represent abstract recipes for the Layer#drc universal DRC function. For example, when using a universal DRC expression like this:

```
out = in.drc(width < 2.0)
```

"width < 2.0" forms a DRC expression object. DRC expression objects have methods which manipulate or evaluate the results of this expression. In addition, DRC expressions have a result type, which is either polygon, edge or edge pair. The result type is defined by the expression generating it. In the example above, "width < 2.0" is a DRC width check which renders edge pairs. To obtain polygons from these edge pairs, use the "polygons" method:

```
out = in.drc((width < 2.0).polygons)
```

The following global functions are relevant for the DRC expressions:

- angle
- area
- area_ratio
- bbox_aspect_ratio
- bbox_height
- bbox_max
- bbox_min
- bbox_width
- corners
- covering
- enc
- enclosing
- extent_refs
- extents
- foreign
- holes
- hulls
- if_all
- if_any
- if_none
- inside

- interacting
- iso
- length
- middle
- notch
- outside
- overlap
- overlapping
- perimeter
- primary
- rectangles
- rectilinear
- relative_height
- rounded_corners
- secondary
- separation
- sep
- sized
- smoothed
- space
- squares
- switch
- width
- with_holes

The following documentation will list the methods available for DRC expression objects.

"!" - Logical not

Usage:

- `! expression`

This operator will evaluate the expression after. If this expression renders an empty result, the operator will return the primary shape. Otherwise it will return an empty result.

This operator can be used together with predicates such a "rectangles" to invert their meaning. For example, this code selects all primary shapes which are not rectangles:

```
out = in.drc(! rectangles)
out = in.drc(! primary.rectangles)   # equivalent
```

"&" - Boolean AND between the results of two expressions

Usage:

- `expression & expression`

The & operator will compute the boolean AND between the results of two expressions. The expression types need to be edge or polygon.

The following example computes the partial edges where width is less than 0.3 micrometers and space is less than 0.2 micrometers:

```
out = in.drc((width < 0.3).edges & (space < 0.2).edges)
```

"+" - Boolean OR between the results of two expressions

Usage:

- `expression + expression`

The + operator will join the results of two expressions.

"-" - Boolean NOT between the results of two expressions

Usage:

- `expression - expression`

The - operator will compute the difference between the results of two expressions. The NOT operation is defined for polygons, edges and polygons subtracted from edges (first argument edge, second argument polygon).

CAUTION: be careful not to take secondary input for the first argument. This will not render the desired results. Remember that the "drc" function will walk over all primary shapes and present single primaries to the NOT operation together with the secondaries of that single shape. So when you use secondary shapes as the first argument, they will not see all all the primaries required to compute the correct result. That's also why a XOR operation cannot be provided in the context of a generic DRC function.

The following example will produce edge markers where the width of is less then 0.3 micron but not inside polygons on the "waive" layer:

```
out = in.drc((width < 0.3).edges - secondary(waive))
```

"angle" - Selects edges based on their angle

Usage:

- `expression.angle (in condition)`

This operation selects edges by their angle, measured against the horizontal axis in the mathematical sense.

For this measurement edges are considered without their direction and straight lines. A horizontal edge has an angle of zero degree. A vertical one has an angle of 90 degrees. The angle range is from -90 (exclusive) to 90 degree (inclusive).

If the input shapes are not polygons or edge pairs, they are converted to edges before the angle test is made.

For example, the following code selects all edges from the primary shape which are 45 degree (up) or 135 degree (down). The "+" will join the results:

```
out = in.drc((angle == 45) + (angle == 135))
out = in.drc((primary.angle == 45) + (primary.angle == 135))    # equivalent
```

Note that angle checks usually imply the need to rotation variant formation as cells which are placed non-rotated and rotated by 90 degree cannot be considered identical. This imposes a performance penalty in hierarchical mode. If possible, consider using DRC#rectilinear for example to detect shapes with non-manhattan geometry instead of using angle checks.

The "angle" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.angle".

"area" - Selects the primary shape if the area is meeting the condition

Usage:

- `expression.area (in condition)`

This operation is used in conditions to select shapes based on their area. It is applicable on polygon expressions. The result will be the input polygons if the area condition is met.

See Layer#drc for more details about comparison specs.

The following example will select all polygons with an area less than 2.0 square micrometers:

```
out = in.drc(area < 2.0)
out = in.drc(primary.area < 2.0)   # equivalent
```

The area method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.area".

"area_ratio" - Selects the input polygon according to its area ratio (bounding box area by polygon area)

Usage:

- `expression.area_ratio (in condition)`

This operation is used in conditions to select shapes based on their area ratio. The area ratio is the ratio of bounding box vs. polygon area. It's a measure how "sparse" the polygons are and how good an approximation the bounding box is. The value is always larger or equal than 1. Boxes have a value of 1.

This filter is applicable on polygon expressions. The result will be the input polygon if the condition is met.

See Layer#drc for more details about comparison specs.

The following example will select all polygons whose area ratio is larger than 3:

```
out = in.drc(area_ratio > 3)
out = in.drc(primary.area_ratio > 3)   # equivalent
```

The "area_ratio" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.area_ratio".

"area_sum" - Selects the input polygons if the sum of all areas meets the condition

Usage:

- `expression.area_sum (in condition)`

Returns the input polygons if the sum of their areas meets the specified condition. This condition is evaluated on the total of all shapes generated in one step of the "drc" loop. As there is a single primary in each loop iteration, "primary.area_sum" is equivalent to "primary.area".

See Layer#drc for more details about comparison specs.

"bbox_aspect_ratio" - Selects the input polygon according to the aspect ratio of the bounding box

Usage:

- `expression.bbox_aspect_ratio (in condition)`

This operation is used in conditions to select shapes based on aspect ratios of their bounding boxes. The aspect ratio is computed by dividing the larger of width and height by the smaller of both. The aspect ratio is always larger or equal to 1. Square or square-boxed shapes have a bounding box aspect ratio of 1.

This filter is applicable on polygon expressions. The result will be the input polygon if the bounding box condition is met.

See Layer#drc for more details about comparison specs.

The following example will select all polygons whose bounding box aspect ratio is larger than 3:

```
out = in.drc(bbox_aspect_ratio > 3)
out = in.drc(primary.bbox_aspect_ratio > 3)   # equivalent
```

The "bbox_aspect_ratio" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.bbox_aspect_ratio".

"bbox_height" - Selects the input polygon if its bounding box height is meeting the condition

Usage:

- `expression.bbox_height (in condition)`

This operation acts similar to DRC#bbox_min, but takes the height of the shape's bounding box. In general, it's more advisable to use DRC#bbox_min or DRC#bbox_max because bbox_height implies a certain orientation. This can imply variant formation in hierarchical contexts: cells rotated by 90 degree have to be treated differently from ones not rotated. This usually results in a larger computation effort and larger result files.

The "bbox_height" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.bbox_height".

"bbox_max" - Selects the input polygon if its bounding box larger dimension is meeting the condition

Usage:

- `expression.bbox_max (in condition)`

This operation acts similar to DRC#bbox_min, but takes the larger dimension of the shape's bounding box.

The "bbox_max" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.bbox_max".

"bbox_min" - Selects the input polygon if its bounding box smaller dimension is meeting the condition

Usage:

- `expression.bbox_min (in condition)`

This operation is used in conditions to select shapes based on smaller dimension of their bounding boxes. It is applicable on polygon expressions. The result will be the input polygons if the bounding box condition is met.

See Layer#drc for more details about comparison specs.

The following example will select all polygons whose bounding box smaller dimension is larger than 200 nm:

```
out = in.drc(bbox_min > 200.nm)
out = in.drc(primary.bbox_min > 200.nm)   # equivalent
```

The "bbox_min" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.bbox_min".

"bbox_width" - Selects the input polygon if its bounding box width is meeting the condition

Usage:

- `expression.bbox_width (in condition)`

This operation acts similar to DRC#bbox_min, but takes the width of the shape's bounding box. In general, it's more advisable to use DRC#bbox_min or DRC#bbox_max because bbox_width implies a certain orientation. This can imply variant formation in hierarchical contexts: cells rotated by 90 degree have to be treated differently from ones not rotated. This usually results in a larger computation effort and larger result files.

The "bbox_width" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.bbox_width".

"centers" - Returns the part at the center of each edge of the input

Usage:

- `expression.centers(length)`
- `expression.end_segments(length, fraction)`

This method acts on edge expressions and delivers a specific part of each edge. See layer#centers for details about this functionality.

"corners" - Selects corners of polygons

Usage:

- `expression.corners`
- `expression.corners(as_dots)`
- `expression.corners(as_boxes)`

This operation acts on polygons and selects the corners of the polygons. It can be put into a condition to select corners by their angles. The angle of a corner is positive for a turn to the left if walking a polygon clockwise and negative for the turn to the right. Hence positive angles indicate concave (inner) corners, negative ones indicate convex (outer) corners. Angles take values between -180 and 180 degree.

When using "as_dots" for the argument, the operation will return single-point edges at the selected corners. With "as_boxes" (the default), small (2x2 DBU) rectangles will be produced at each selected corner.

The following example selects all corners:

```
out = in.drc(corners)
out = in.drc(primary.corners)    # equivalent
```

The following example selects all inner corners:

```
out = in.drc(corners > 0)
out = in.drc(primary.corners > 0)    # equivalent
```

The "corners" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.corners".

"count" - Selects a expression result based on the number of (local) shapes

Usage:

- `expression.count (in condition)`

This operation is used in conditions to select expression results based on their count. "count" is used as a method on a expression. It will evaluate the expression locally and return the original result if the shape count in the result is matching the condition.

See Layer#drc for more details about comparison specs.

Note that the expression is evaluated locally: for each primary shape, the expression is evaluated and the count of the resulting edge, edge pair or polygon set is taken. As the primary input will always have a single item (the local shape), using "count" on primary does not really make sense. It can be used on derived expressions however.

The following example selects triangles:

```
out = in.drc(if_any(corners.count == 3))
```

Note "if_any" which selects the primary shape if the argument evaluates to a non-empty result. Without "if_any" three corners are returned for each triangle.

"covering" - Selects shapes entirely covering other shapes

Usage:

- `expression.covering(other) (optionally in conditions)`
- `covering(other) (optionally in conditions)`

This method represents the selector of primary shapes which entirely cover shapes from the other layer. This version can be put into a condition indicating how many shapes of the other layer need to be covered. Use this variant within DRC expressions (also see Layer#drc).

For example, the following statement selects all input shapes which entirely cover shapes from the "other" layer:

```
out = in.drc(covering(other))
```

This example selects all input shapes which entire cover shapes from the other layer and there are more than two shapes from "other" inside primary shapes:

```
out = in.drc(covering(other) > 2)
```

"edges" - Converts the input shapes into edges

Usage:

- `expression.edges`

- `expression.edges(mode)`

Polygons will be separated into edges forming their contours. Edge pairs will be decomposed into individual edges.

Contrary most other operations, "edges" does not have a plain function equivalent as this is reserved for the function generating an edges layer. To generate the edges of the primary shapes, use "primary" explicit as the source for the edges:

```
out = in.drc(primary.edges)
```

The "mode" argument allows selecting specific edges from polygons. Allowed values are: "convex", "concave", "step", "step_in" and "step_out". "step" generates edges only if they provide a step between two other edges. "step_in" creates edges that make a step towards the inside of the polygon and "step_out" creates edges that make a step towards the outside:

```
out = in.drc(primary.edges(convex))
```

In addition, "not_.." variants are available which selects edges not qualifying for the specific mode:

```
out = in.drc(primary.edges(not_convex))
```

The mode argument is ignored when translating other objects than polygons.

"end_segments" - Returns the part at the end of each edge of the input

Usage:

- `expression.end_segments(length)`
- `expression.end_segments(length, fraction)`

This method acts on edge expressions and delivers a specific part of each edge. See layer#end_segments for details about this functionality.

"extended" - Returns polygons describing an area along the edges of the input

Usage:

- `expression.extended([:begin => b,] [:end => e,] [:out => o,] [:in => i], [:joined => true])`
- `expression.extended(b, e, o, i)`

This method acts on edge expressions. It will create a polygon for each edge tracing the edge with certain offsets to the edge. "o" is the offset applied to the outer side of the edge, "i" is the offset applied to the inner side of the edge. "b" is the offset applied at the beginning and "e" is the offset applied at the end.

"extended_in" - Returns polygons describing an area along the edges of the input

Usage:

- `expression.extended_in(d)`

This method acts on edge expressions. Polygons are generated for each edge describing the edge drawn with a certain width extending into the "inside" (the right side when looking from start to end). This method is basically equivalent to the extended method: "extended(0, 0, 0, dist)". A version extending to the outside is extended_out.

"extended_out" - Returns polygons describing an area along the edges of the input

Usage:

- `expression.extended_out(d)`

This method acts on edge expressions. Polygons are generated for each edge describing the edge drawn with a certain width extending into the "outside" (the left side when looking from start to end). This method is basically equivalent to the extended method: "extended(0, 0, dist, 0)". A version extending to the inside is extended_in.

"extent_refs" - Returns partial references to the bounding boxes of the polygons

Usage:

- `expression.extent_refs([ options ])`

The extent_refs operation acts on polygons and has the same effect than Layer#extent_refs. It takes the same arguments. It is available as a method on DRC expressions or as plain function, in which case it acts on the primary shapes.

"extents" - Returns the bounding box of each input object

Usage:

- `expression.extents([ enlargement ])`

This method provides the same functionality as Layer#extents and takes the same arguments. It returns the bounding boxes of the input objects. It acts on edge edge pair and polygon expressions.

The "extents" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.extents".

"first_edges" - Returns the first edges of edge pairs

Usage:

- `expression.first_edges`

This method acts on edge pair expressions and returns the first edges of the edge pairs delivered by the expression.

Some checks deliver symmetric edge pairs (e.g. space, width, etc.) for which the edges are commutable. "first_edges" will deliver both edges for such edge pairs.

"holes" - Selects all holes from the input polygons

Usage:

- `expression.holes`

This operation can be used as a plain function in which case it acts on primary shapes or can be used as method on another DRC expression. The following example selects all holes with an area larger than 2 square micrometers:

```
out = in.drc(holes.area > 2.um)
out = in.drc(primary.holes.area > 2.um)   # equivalent
```

"hulls" - Selects all hulls from the input polygons

Usage:

- `expression.hulls`

The hulls are the outer contours of the input polygons. By selecting hulls only, all holes will be closed.

This operation can be used as a plain function in which case it acts on primary shapes or can be used as method on another DRC expression. The following example closes all holes:

```
out = in.drc(hulls)
out = in.drc(primary.hulls)   # equivalent
```

"inside" - Selects shapes entirely inside other shapes

Usage:

- `expression.inside(other)`
- `inside(other)`

This method represents the selector of primary shapes which are entirely inside shapes from the other layer. Use this variant within DRC expressions (also see Layer#drc).

"interacting" - Selects shapes interacting with other shapes

Usage:

- `expression.interacting(other) (optionally in conditions)`
- `interacting(other) (optionally in conditions)`

See covering for a description of the use cases for this function. When using "interacting", shapes are selected when the interact (overlap, touch) shapes from the other layer.

When using this method with a count, the operation may not render the correct results if the other input is not merged. By nature of the generic DRC feature, only those shapes that interact with the primary shape will be selected. If the other input is split into multiple polygons, not all components may be captured and the computed interaction count may be incorrect.

"length" - Selects edges based on their length

Usage:

- `expression.length (in condition)`

This operation will select those edges which are meeting the length criterion. Non-edge shapes (polygons, edge pairs) will be converted to edges before.

For example, this code selects all edges from the primary shape which are longer or equal than 1 micrometer:

```
out = in.drc(length >= 1.um)
out = in.drc(primary.length >= 1.um)   # equivalent
```

The "length" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.length".

"length_sum" - Selects the input edges if the sum of their lengths meets the condition

Usage:

- `expression.length_sum (in condition)`

Returns the input edges if the sum of their lengths meets the specified condition. This condition is evaluated on the total of all edges generated in one step of the "drc" loop.

See Layer#drc for more details about comparison specs.

"merged" - Returns the merged input polygons, optionally selecting multi-overlap

Usage:

- `expression.merged`
- `expression.merged(min_count)`

This operation will act on polygons. Without a min_count argument, the merged polygons will be returned.

With a min_count argument, the result will include only those parts where more than the given number of polygons overlap. As the primary input is merged already, it will always contribute as one.

The "merged" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.merged".

"middle" - Returns the centers of polygon bounding boxes

Usage:

- `expression.middle([ options ])`

The middle operation acts on polygons and has the same effect than Layer#middle. It takes the same arguments. It is available as a method on DRC expressions or as plain function, in which case it acts on the primary shapes.

"outside" - Selects shapes entirely outside other shapes

Usage:

- `expression.outside(other)`
- `outside(other)`

This method represents the selector of primary shapes which are entirely outside shapes from the other layer. Use this variant within DRC expressions (also see Layer#drc).

"overlapping" - Selects shapes overlapping with other shapes

Usage:

- `expression.overlapping(other) (optionally in conditions)`
- `overlapping(other) (optionally in conditions)`

See covering for a description of the use cases for this function. When using "overlapping", shapes are selected when the overlap shapes from the other layer.

When using this method with a count, the operation may not render the correct results if the other input is not merged. By nature of the generic DRC feature, only those shapes that interact with the primary shape will be selected. If the other input is split into multiple polygons, not all components may be captured and the computed interaction count may be incorrect.

"perimeter" - Selects the input polygon if the perimeter is meeting the condition

Usage:

- `expression.perimeter (in condition)`

This operation is used in conditions to select shapes based on their perimeter. It is applicable on polygon expressions. The result will be the input polygons if the perimeter condition is met.

See Layer#drc for more details about comparison specs.

The following example will select all polygons with a perimeter less than 10 micrometers:

```
out = in.drc(perimeter < 10.0)
out = in.drc(primary.perimeter < 10.0)   # equivalent
```

The perimeter method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.perimeter".

"perimeter_sum" - Selects the input polygons if the sum of all perimeters meets the condition

Usage:

- `expression.perimeter_sum (in condition)`

Returns the input polygons if the sum of their perimeters meets the specified condition. This condition is evaluated on the total of all shapes generated in one step of the "drc" loop. As there is a single primary in each loop iteration, "primary.perimeter_sum" is equivalent to "primary.perimeter".

See Layer#drc for more details about comparison specs.

"polygons" - Converts the input shapes into polygons

Usage:

- `expression.polygons([ enlargement ])`

Generates polygons from the input shapes. Polygons stay polygons. Edges and edge pairs are converted to polygons. For this, the enlargement parameter will specify the edge thickness or augmentation applied to edge pairs. With the default enlargement of zero edges will not be converted to valid polygons and degenerated edge pairs will not become valid polygons as well.

Contrary most other operations, "polygons" does not have a plain function equivalent as this is reserved for the function generating a polygon layer.

This method is useful for generating polygons from DRC violation markers as shown in the following example:

```
out = in.drc((width < 0.5.um).polygons)
```

"rectangles" - Selects all polygons which are rectangles

Usage:

- `expression.rectangles`

This operation can be used as a plain function in which case it acts on primary shapes or can be used as method on another DRC expression. The following example selects all rectangles:

```
out = in.drc(rectangles)
out = in.drc(primary.rectangles)    # equivalent
```

"rectilinear" - Selects all polygons which are rectilinear

Usage:

- `expression.rectilinear`

Rectilinear polygons only have vertical and horizontal edges. Such polygons are also called manhattan polygons.

This operation can be used as a plain function in which case it acts on primary shapes or can be used as method on another DRC expression. The following example selects all manhattan polygons:

```
out = in.drc(rectilinear)
out = in.drc(primary.rectilinear)    # equivalent
```

"relative_height" - Selects the input polygon according to the height vs. width of the bounding box

Usage:

- `expression.relative_height (in condition)`

This operation is used in conditions to select shapes based on the ratio of bounding box height vs. width. The taller the shape, the larger the value. Wide polygons have a value below 1. A square has a relative height of 1.

This filter is applicable on polygon expressions. The result will be the input polygon if the condition is met.

Don't use this method if you can use bbox_aspect_ratio, because the latter is isotropic and can be used hierarchically without generating rotation variants.

See Layer#drc for more details about comparison specs.

The following example will select all polygons whose relative height is larger than 3:

```
out = in.drc(relative_height > 3)
out = in.drc(primary.relative_height > 3)   # equivalent
```

The "relative_height" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.bbox_aspect_ratio".

"rounded_corners" - Applies corner rounding

Usage:

- `expression.rounded_corners(inner, outer, n)`

This operation acts on polygons and applies corner rounding to the given inner and outer corner radius and the number of points n per full circle. See Layer#rounded_corners for more details.

The "rounded_corners" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.rounded_corners".

"second_edges" - Returns the second edges of edge pairs

Usage:

- `expression.second_edges`

This method acts on edge pair expressions and returns the second edges of the edge pairs delivered by the expression.

Some checks deliver symmetric edge pairs (e.g. space, width, etc.) for which the edges are commutable. "second_edges" will not deliver edges for such edge pairs. Instead, "first_edges" will deliver both.

"sized" - Returns the sized version of the input

Usage:

- `expression.sized(d [, mode])`
- `expression.sized(dx, dy [, mode]))`

This method provides the same functionality as Layer#sized and takes the same arguments. It acts on polygon expressions.

The "sized" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.sized".

"smoothed" - Applies smoothing

Usage:

- `expression.smoothed(d [, keep_hv ])`

This operation acts on polygons and applies polygon smoothing with the tolerance d. 'keep_hv' indicates whether horizontal and vertical edges are maintained. Default is 'no' which means such edges may be distorted. See Layer#smoothed for more details.

The "smoothed" method is available as a plain function or as a method on DRC expressions. The plain function is equivalent to "primary.smoothed".

"squares" - Selects all polygons which are squares

Usage:

- `expression.squares`

This operation can be used as a plain function in which case it acts on primary shapes or can be used as method on another DRC expression. The following example selects all squares:

```
out = in.drc(squares)
out = in.drc(primary.squares)    # equivalent
```

"start_segments" - Returns the part at the beginning of each edge of the input

Usage:

- `expression.start_segments(length)`
- `expression.start_segments(length, fraction)`

This method acts on edge expressions and delivers a specific part of each edge. See layer#start_segments for details about this functionality.

"with_holes" - Selects all input polygons with the specified number of holes

Usage:

- `expression.with_holes (in condition)`

This operation can be used as a plain function in which case it acts on primary shapes or can be used as method on another DRC expression. The following example selects all polygons with more than 2 holes:

```
out = in.drc(with_holes > 2)
out = in.drc(primary.with_holes > 2)   # equivalent
```

"|" - Boolean OR between the results of two expressions

Usage:

- `expression | expression`

The | operator will compute the boolean OR between the results of two expressions. '+' is basically a synonym. Both expressions must render the same type.

2.20.2. DRC Reference: Layer Object

"&" - Boolean AND operation

Usage:

- `self & other`

The method computes a boolean AND between self and other.

This method is available for polygon and edge layers. An alias is "and". See there for a description of the function.

"+" - Join layers

Usage:

- `self + other`

The method includes the edges or polygons from the other layer into this layer. The "+" operator is an alias for the join method.

This method is available for polygon, edge and edge pair layers. An alias is "join". See there for a description of the function.

"-" - Boolean NOT operation

Usage:

- `self - other`

The method computes a boolean NOT between self and other.

This method is available for polygon and edge layers. An alias is "not". See there for a description of the function.

"^" - Boolean XOR operation

Usage:

- `self ^ other`

The method computes a boolean XOR between self and other.

This method is available for polygon and edge layers. An alias is "xor". See there for a description of the function.

"and" - Boolean AND operation

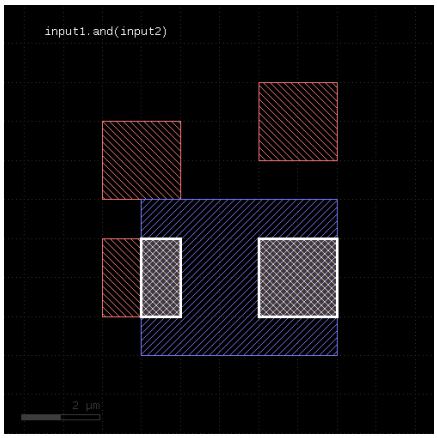
Usage:

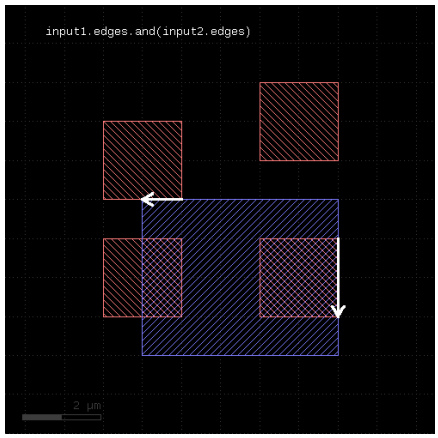
- `layer.and(other [, prop_constraint ])`

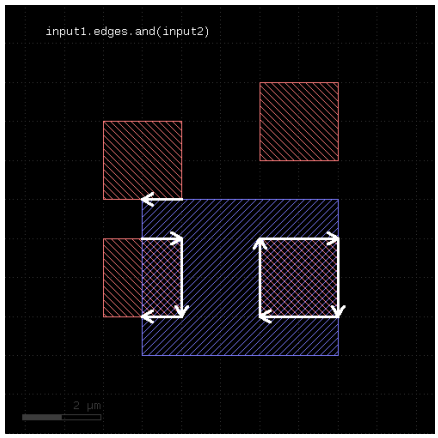
The method computes a boolean AND between self and other. It is an alias for the "&" operator which lacks the ability to specify a properties constraint.

This method is available for polygon and edge layers. If the first operand is an edge layer and the second is a polygon layer, the result will be the edges of the first operand which are inside or on the borders of the polygons of the second operand.

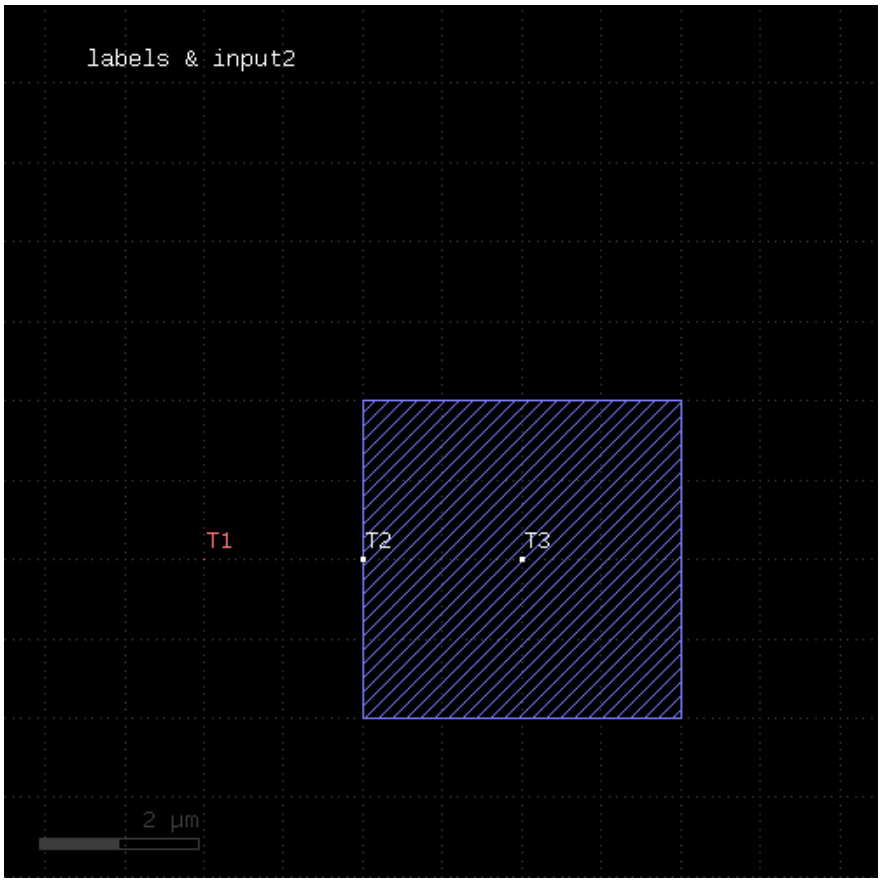
The following images show the effect of the "and" method on polygons and edges (input1: red, input2: blue):







The AND operation can be applied between a text and a polygon layer. In this case, the texts inside or at the border of the polygons will be written to the output (labels: red, input2: blue):



When a properties constraint is given, the operation is performed only between shapes with the given relation. Together with the ability to provide net-annotated shapes through the nets method, this allows constraining the boolean operation to shapes from the same or from different nets.

See prop_eq, prop_ne and prop_copy for details.

"andnot" - Computes Boolean AND and NOT results at the same time

Usage:

- `layer.andnot(other)`

This method returns a two-element array containing one layer for the AND result and one for the NOT result.

This method is available for polygon and edge layers. For polygon layers, the other input must be a polygon layer too. For edge layers, the other input can be polygon or edge.

It can be used to initialize two variables with the AND and NOT results:

```
(and_result, not_result) = l1.andnot(l2)
```

As the AND and NOT results are computed in the same sweep, calling this method is faster than calling AND and NOT separately.

"area" - Returns the total area of the polygons in the region

Usage:

- `layer.area`

This method requires a polygon layer. It returns the total area of all polygons in square micron. Merged semantics applies, i.e. before computing the area, the polygons are merged unless raw mode is chosen (see raw). Hence, in clean mode, overlapping polygons are not counted twice.

The returned value gives the area in square micrometer units.

"bbox" - Returns the overall bounding box of the layer

Usage:

- `layer.bbox`

The return value is a DBox object giving the bounding box in micrometer units.

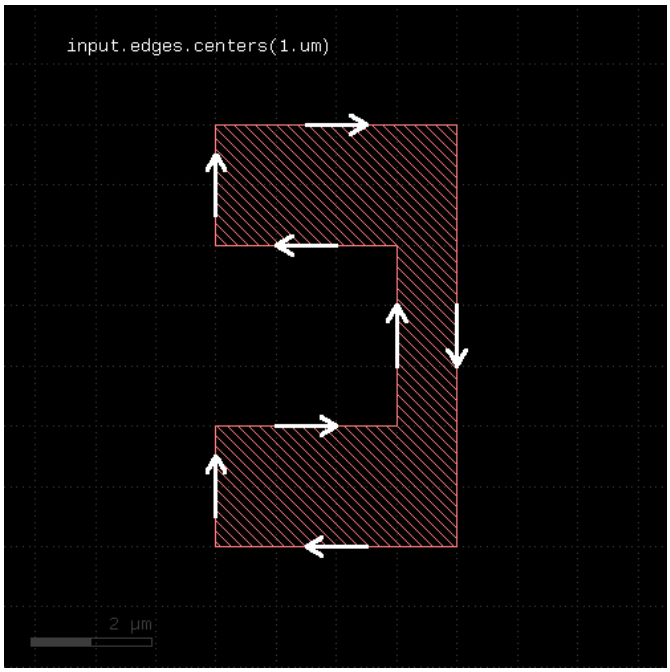
"centers" - Returns the center parts of the edges

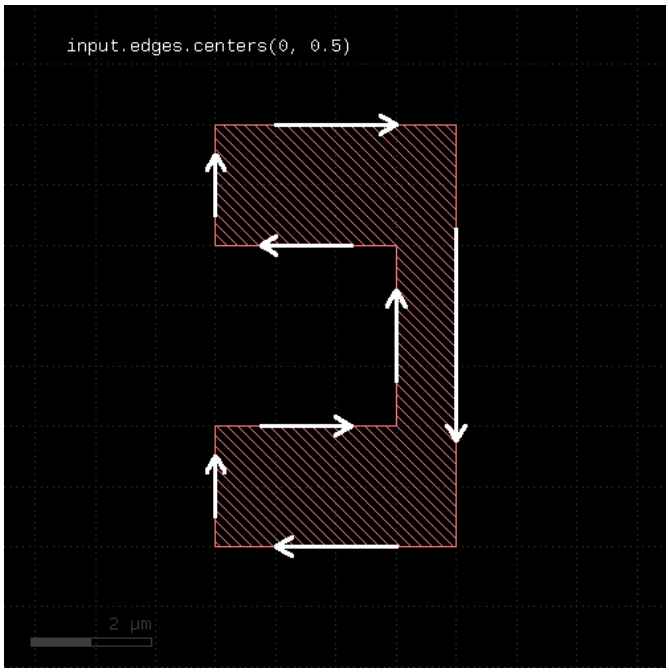
Usage:

- `layer.centers(length)`
- `layer.centers(length, fraction)`

Similar to start_segments and end_segments, this method will return partial edges for each given edge in the input. For the description of the parameters see start_segments or end_segments.

The following images show the effect of the method:





"clean" - Marks a layer as clean

Usage:

- `layer.clean`

A layer marked as clean will provide "merged" semantics, i.e. overlapping or touching polygons are considered as single polygons. Inner edges are removed and collinear edges are connected. Clean state is the default.

See raw for some remarks about how this state is propagated.

"collect" - Transforms a layer

Usage:

- `layer.collect { |object| ... }`

This method evaluates the block for each object in the layer and returns a new layer with the objects returned from the block. It is available for edge, polygon and edge pair layers. The corresponding objects are DPolygon, DEdge or DEdgePair.

If the block evaluates to nil, no object is added to the output layer. If it returns an array, each of the objects in the array is added. The returned layer is of the original type and will only accept objects of the respective type. Hence, for polygon layers, DPolygon objects need to be returned. For edge layers those need to be DEdge and for edge pair layers, they need to be DEdgePair objects. For convenience, Polygon, Edge and EdgePair objects are accepted too and are scaled by the database unit to render micrometer-unit objects. Region, Edges and EdgePair objects are accepted as well and the corresponding content of that collections is inserted into the output layer.

Other versions are available that allow translation of objects into other types (collect_to_region, collect_to_edges and collect_to_edge_pairs).

Because this method executes inside the interpreter, it's inherently slow. Tiling does not apply to this method.

Here is a slow equivalent of the rotated method

```
# Rotates by 45 degree
t = DCplxTrans(1.0, 45.0, false, DVector::new)
```

```
new_layer = layer.collect { |polygon| polygon.transformed(t) }
```

"collect_to_edge_pairs" - Transforms a layer into edge pair objects

Usage:

- `layer.collect_to_edge_pairs { |object| ... }`

This method is similar to collect, but creates an edge pair layer. It expects the block to deliver EdgePair, DEdgePair or EdgePairs objects.

"collect_to_edges" - Transforms a layer into edge objects

Usage:

- `layer.collect_to_edges { |object| ... }`

This method is similar to collect, but creates an edge layer. It expects the block to deliver objects that can be converted into edges. If polygon-like objects are returned, their contours will be turned into edge sequences.

"collect_to_region" - Transforms a layer into polygon objects

Usage:

- `layer.collect_to_region { |object| ... }`

This method is similar to collect, but creates a polygon layer. It expects the block to deliver objects that can be converted into polygons. Such objects are of class DPolygon, DBox, DPath, Polygon, Path, Box and Region.

"corners" - Selects corners of polygons

Usage:

- `layer.corners([ options ])`
- `layer.corners(angle [, options ])`
- `layer.corners(amin .. amax [, options ])`

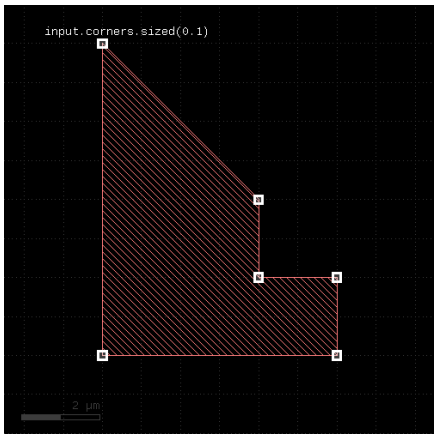
This method produces markers on the corners of the polygons. An angle criterion can be given which selects corners based on the angle of the connecting edges. Positive angles indicate a left turn while negative angles indicate a right turn. Since polygons are oriented clockwise, positive angles indicate concave (inner) corners while negative ones indicate convex (outer) corners

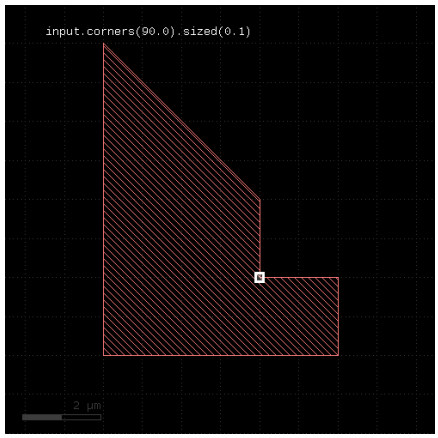
The markers generated can be point-like edges or small 2x2 DBU boxes. The latter is the default.

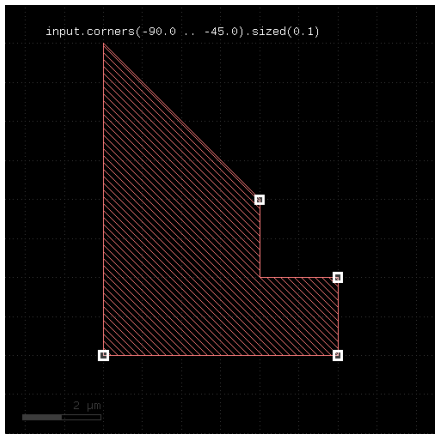
The options available are:

- **as_boxes** : with this option, small boxes will be produced as markers
- **as_dots** : with this option, point-like edges will be produced instead of small boxes
- **as_edge_pairs** : with this option, an edge pair is produced for each corner selected. The first edge is the incoming edge to the corner, the second edge the outgoing edge.

The following images show the effect of this method:







"count" - Returns the number of objects on the layer

Usage:

- `layer.count`

The count is the number of raw objects, not merged regions or edges. This is the flat count - the number of polygons, edges or edge pairs seen from the top cell. "count" can be computationally expensive for original layers with clip regions or cell tree filters.

See hier_count for a hierarchical (each cell counts once) count.

"covering" - Selects shapes or regions of self which completely cover (enclose) one or more shapes from the other region

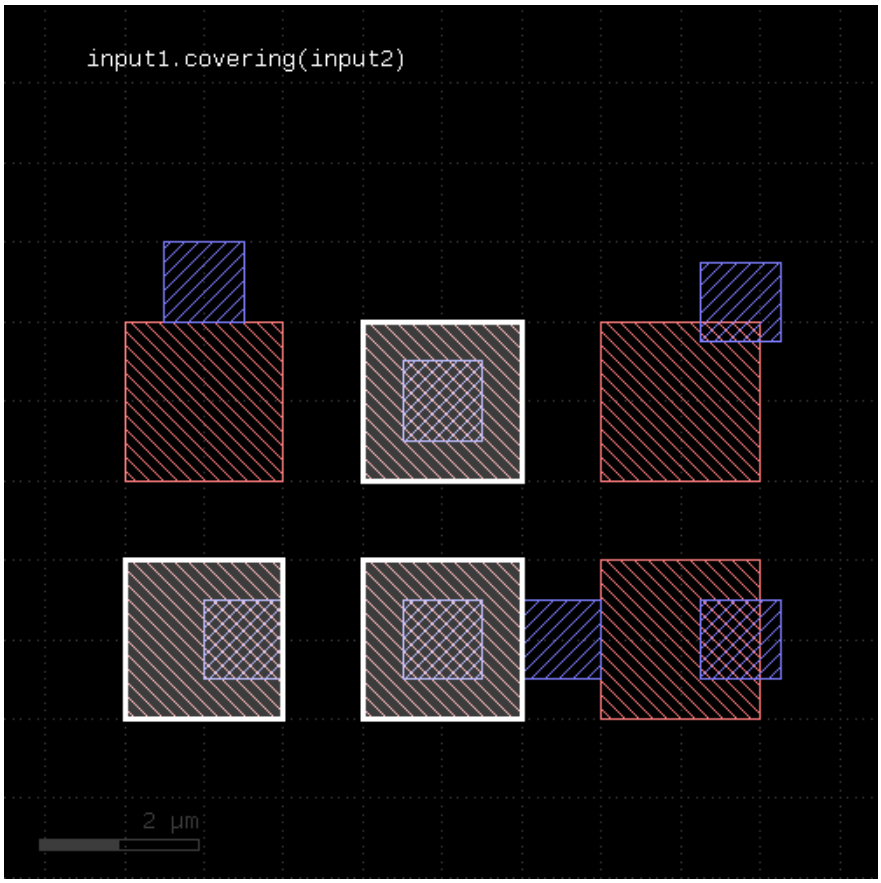
Usage:

- `layer.covering(other)`
- `layer.covering(other, min_count)`
- `layer.covering(other, min_count, max_count)`
- `layer.covering(other, min_count .. max_count)`

This method selects all shapes or regions from self which completly cover shapes from the other region. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. It returns a new layer containing the selected shapes. A version which modifies self is select_covering.

This method is available for polygons only.

The following image shows the effect of the "covering" method:



A range of counts can be specified. If so, the shape from the primary layer is only selected when covering a given number of shapes from the other layer. For the interpretation of the count see interacting.

The "covering" attribute is sometimes called "enclosing", but this name is used for the respective DRC function (see enclosing).

"data" - Gets the low-level data object

Usage:

- `layer.data`

This method returns a Region, Edges or EdgePairs object representing the underlying RBA object for the data. Access to these objects is provided to support low-level iteration and manipulation of the layer's data.

"drc" - Provides a generic DRC function for use with DRC expressions

Usage:

- `layer.drc(expression [, prop_constraint ])`

This method implements the universal DRC which offers enhanced abilities, improved performance in some applications and better readability.

The key concept for this method are DRC expressions. DRC expressions are formed by using predefined keywords like "width", operators like "&" and methods to build an abstract definition of the operations to perform within the DRC.

When the DRC function is executed, it will basically visit all shapes from the input layer (the layer, the "drc" method is called on)). While it does, it collects the neighbor shapes from all involved other inputs and runs the requested operations on each cluster. Currently, "drc" is only available for polygon layers.

This way, the nature of the "drc" operation is that of the loop over all (merged) input polygons. Within the operation executed on each shape, it's possible to make decisions such as "if the shape has an area larger than something, apply this operation" or similar. This often can be achieved with conventional DRC functions too, but involves potentially complex and heavy operations such as booleans, interact etc. For this reason, the "drc" function may provide a better performance.

In addition, within the loop a single shape from the input layer is presented to the execution engine which runs the operations. This allows using operations such as "size" without having to consider neighbor polygons growing into the area of the initial shape. In this sense, the "drc" function sees the layer as individual polygons rather than a global "sea of polygons". This enables new applications which are otherwise difficult to implement.

Primaries and secondaries

An important concept in "drc" expressions is the "primary". The primary represents a single shape from the input layer. "Secondaries" are shapes from other inputs. Primaries guide the operation - secondaries without primaries are not seen. The "drc" operation will look for secondaries within a certain distance which is determined from the operations from the expression to execute. The secondaries collected in this step will not be merged, so the secondary polygons may be partial. This is important when using measurement operations like "area" on secondary polygons.

Checks

Here is an example for a generic DRC operation which performs a width check for less than 0.5um on the primary shapes. It uses the "width" operator:

```
out = in.drc(width < 0.5.um)
```

Other single or double-bounded conditions are available too, for example:

```
out = in.drc(width <= 0.5.um)
out = in.drc(width > 0.5.um)
out = in.drc(width == 0.5.um)
out = in.drc(width != 0.5.um)
out = in.drc(0.2.um < width < 0.5.um)
```

To specify the second input for a two-layer check, add it to the check function. This example shows how to use a two-layer separation check ("separation"):

```
l1 = input(1, 0)
l2 = input(2, 0)
out = l1.drc(separation(l2) < 0.5.um)
```

The second input of this check function can be a computed expression. In this case the local loop will first evaluate the expression for the second input and then use the result as second input in the check. Note that this computation is performed locally and separately for each primary and its context.

Options for the checks are also specified inside the brackets. For example, to select projection metrics ("projection") for the "width" check use:

```
out = in.drc(width(projection) < 0.5.um)
```

Edges and edge pairs

Although the "drc" function operates on polygon layers, internally it is able to handle edge and edge pair types too. Some operations generate edge pairs, some other generate edges. As results from one operation can be processed further in the DRC expressions, methods are available to filter, process and convert these types.

For example, all checks produce edge pairs which can be converted into polygons using the "polygons" method:

```
out = in.drc((width(projection) < 0.5.um).polygons)
```

Note a subtle detail: when putting the "polygons" method inside the "drc" brackets, it is executed locally on every visited primary polygon. The result in this case is identical to the global conversion:

```
# same, but with "global" conversion:
out = in.drc(width(projection) < 0.5.um).polygons
```

But having the check polygons inside the loop opens new opportunities and is more efficient in general. In the previous example, the local conversion will keep a few edge pairs after having converted them to polygons. In the global case, all edge pairs are collected first and then converted. If there are many edge pairs, this requires more memory and a larger computing overhead for managing the bigger number of shapes.

For the conversion of edges, edge pairs and polygons into other types, these methods are provided:

- "DRC#polygons": converts edge pairs to polygons
- "DRC#extended", "DRC#extended_in", "DRC#extended_out": converts edges to polygons
- "DRC#first_edges", DRC#second_edges": extracts edges from edge pairs
- "DRC#edges": decomposes edge pairs and polygons into edges
- "DRC#corners": can extract corners from polygons

The following example decomposes the primary polygons into edges:

```
out = in.drc(primary.edges)
```

(for backward compatibility you cannot abbreviate "primary.edges" simply as "edges" like other functions).

The previous example isn't quite exciting as it is equivalent to

```
# Same as above
out = in.edges
```

But it gets more interesting, as within the loop, "edges" delivers the edge set for each individual polygon. It's possible to work with this distinct set, so for example this will give you the edges of polygons with more than four corners:

```
out = in.drc(primary.edges.count > 4)
```

Explanation: "count" is a "quantifier" which takes any kind of set (edges, edge pairs, polygons) and returns the set if the number of inhabitants meets the given condition. Otherwise the set is skipped. So it will look at the edges and if there are more than four (per primary shape), it will forward this set.

The same result can be achieved with classic DRC with "interact" and a figure count, but at a much higher computation cost.

Edge and edge/polygon operations

The "drc" framework supports the following edge and edge/polygon operations:

- Edge vs. edge and edge vs. polygon booleans
- Edge vs. polygon interactions ("DRC#interacting", "DRC#overlapping")
- Edge sampling ("DRC#start_segments", "DRC#centers", "DRC#end_segments")

Filters

Filter operators select input polygons or edges based on their properties. These filters are provided:

- "DRC#area": selects polygons based on their area
- "DRC#perimeter": selects polygons based on their perimeter
- "DRC#area_ratio": selects polygons based on their bounding box to polygon area ratio
- "DRC#bbox_aspect_ratio": selects polygons based on their bounding box aspect ratio
- "DRC#relative_height": selects polygons based on their relative height
- "DRC#bbox_min", "DRC#bbox_max", "DRC#bbox_width", "DRC#bbox_height": selects polygons based on their bounding box properties
- "DRC#length": selects edges based on their length
- "DRC#angle": selects edges based on their orientation

For example, to select polygons with an area larger than one square micrometer, use:

```
out = in.drc(area > 1.0)
```

For the condition, use the usual numerical bounds like:

```
out = in.drc(area == 1.0)
out = in.drc(area <= 1.0)
out = in.drc(0.2 < area < 1.0)
```

The result of the area operation is the input polygon if the area condition is met.

In the same fashion, "perimeter" applies to the perimeter of the polygon. "bbox_min" etc. will evaluate a particular dimensions of the polygon's bounding box and use the respective dimension for filtering the polygon.

Note that it's basically possible to use the polygon filters on any input - computed and secondaries. In fact, plain "area" for example is a shortcut for "primary.area" indicating that the area of primary shapes are supposed to be computed. However, any input other than the primary is not necessarily complete or it may consist of multiple polygons. Hence the computed values may be too big or too small. It's recommended therefore to use the measurement functions on primary polygons unless you know what you're doing.

Filter predicates

The "drc" feature also supports some predicates. "predicates" are boolean values indicating a certain condition. A predicate filter works in a way that it only passes the polygons if the condition is met.

The predicates available currently are:

- "DRC#rectangles": Filters rectangles

- "DRC#squares": Filters squares
- "DRC#rectilinear": Filters rectilinear ("Manhattan") polygons

For the same reason as explained above, it's recommended to use these predicates standalone, so they act on primary shapes. It's possible to use the predicates on computed shapes or secondaries, but that may not render the desired results.

Logical NOT operator

The "!" operator will evaluate the expression behind it and return the current primary shape if the input is empty and return an empty polygon set if not. Hence the following filter will deliver all polygons which are not rectangles:

```
out = in.drc(! rectangles)
```

Logical combination operators

The logical "if_any" or "if_all" functions allow connecting multiple conditions and evaluate to "true" (means: a non-empty shape set) if either one input is a non-empty shape set ("if_any") or if all inputs are non-empty ("if_all").

For example, this will select all polygons which are rectangles and whose area is larger than 20 square micrometers:

```
out = in.drc(if_all(rectangles, area > 20.0))
```

"if_all" delivers the primary shape if all of the input expressions render a non-empty result.

In contrast to this, the "if_any" operation will deliver the primary shape if one of the input expressions renders a non-empty result.

The "switch" function allows selecting one input based on the results of an expression. In the two-input form it's equivalent to "if". The first expression is the condition. If it evaluates to a non-empty shape set, the result of the second expression is taken. Otherwise, the result is empty.

Hence the following code delivers all rectangles sized by 100 nm. All other shapes are skipped:

```
out = in.drc(switch(rectangles, primary.sized(100.nm)))
```

A third expression will be considered the "else" branch: the result of this expression will be taken if the first one is not taken. So this example will size all rectangles and leave other shapes untouched:

```
out = in.drc(switch(rectangles, primary.sized(100.nm), primary))
```

If more expressions are given, they are considered as a sequence of condition/result chain (c1, e1, c2, e2, ...) in the sense of "if(c1) return(e1) else if(c2) return(e2) ...". So the e1 is taken if c1 is met, e2 is taken when c1 is not met, but c2 is and so forth. If there is an odd number of expressions, the last one will be the default expression which is taken if none of the conditions is met.

Polygon manipulations

The "drc" operations feature polygon manipulations where the input is either the primary, secondaries or derived shapes. Manipulations include sizing ("sized"), corner rounding ("rounded_corners"), smoothing ("smoothed") and boolean operations.

This example computes a boolean AND between two layers before selecting the result polygons with an area larger than 1 square micrometer. Note that "primary" is a placeholder for the primary shape:

```
l1 = input(1, 0)
l2 = input(2, 0)
```

```
out = l1.drc((primary & l2).area > 1.0)
```

This example demonstrates how the "drc" operation can improve performance: as the boolean operation is computed locally and the result is discarded when no longer required, less shapes need to be stored hence reducing the memory overhead and CPU time required to manage these shapes.

Note that the precise form of the example above is

```
out = l1.drc((primary & secondary(l2)).area > 1.0)
```

The "secondary" operator indicates that "l2" is to be used as secondary input to the "drc" function. Only in this form, the operators of the boolean AND can be reversed:

```
out = l1.drc((secondary(l2) & primary).area > 1.0)
```

Quantifiers

Some filters operate on properties of the full, local, per-primary shape set. While the loop is executed, the DRC expressions will collect shapes, either from the primary, its neighborhood (secondaries) or from deriving shape sets.

Obviously the primary is a simple one: it consists of a single shape, because this is how the loop operates. Derived shape sets however can be more complex. "Quantifiers" allow assessing properties of the complete, per-primary shape set. A simple one is "DRC#count" which checks if the number of shapes within a shape set is within a given range.

Obviously, "primary.count == 1" is always true. So using "count" primaries isn't much fun. So it's better to use it on derived sets. The following condition will select all primary shapes which have more than 13 corners:

```
out = in.drc(if_any(primary.corners.count > 13))
```

Note an important detail here: the "if_any" function will make this statement render primary polygons, if the expression inside gives a non-empty result. Without "if_any", the result would be the output of "count" which is the set of all corners where the corner count is larger than 13.

Expressions as objects

The expression inside the "drc" function is a Ruby object and can be stored in variables. If you need the same expression multiple times, it can be more efficient to use the same Ruby object. In this example, the same expression is used two times. Hence it's computed two times:

```
out = l1.drc(((primary & l2).area == 1.0) + ((primary & l2).area == 2.0))
```

A more efficient version is:

```
overlap_area = (primary & l2).area
out = l1.drc((overlap_area == 1.0) + (overlap_area == 2.0))
```

Note that the first line prepares the operation, but does not execute the area computation or the boolean operation. But when the "drc" function executes the loop over the primaries it will only compute the area once per primary as it is represented by the same Ruby object.

Properties constraints

The method can be given a properties constraint so that it is only performed between shapes with the same or different user properties. Note that properties have to be enabled or generated (e.g. through the DRCLayer#nets method) before they can be used.

Example:

```
connect(metal1, via1)
...

space_not_connected = metal1.nets.drc(space < 0.4.um, props_ne)
```

See props_eq, props_ne and props_copy for details.

Outlook

DRC expressions are quite rich and powerful. They provide a more intuitive way of writing DRC expressions, are more efficient and open new opportunities. DRC development is likely to focus on this scheme in the future.

More formal details about the bits and pieces can be found in the "DRC" class documentation.

"dup" - Duplicates a layer

Usage:

- `layer.dup`

Duplicates the layer. This basically will create a copy and modifications of the original layer will not affect the duplicate. Please note that just assigning the layer to another variable will not create a copy but rather a pointer to the original layer. Hence modifications will then be visible on the original and derived layer. Using the dup method will avoid that.

However, dup will double the memory required to hold the data and performing the deep copy may be expensive in terms of CPU time.

"each" - Iterates over the objects from the layer

Usage:

- `layer.each { |object| ... }`

This method evaluates the block on each object of the layer. Depending on the layer type, these objects are of DPolygon, DEdge or DEdgePair type.

Because this method executes inside the interpreter, it's inherently slow. Tiling does not apply to this method.

"edge_pairs?" - Returns true, if the layer is an edge pair collection

Usage:

- `layer.edge_pairs?`

"edges" - Decomposes the layer into single edges

Usage:

- `layer.edges`
- `layer.edges(mode)`

Edge pair collections are decomposed into the individual edges that make up the edge pairs. Polygon layers are decomposed into the edges making up the polygons. This method returns an edge layer but will not modify the layer it is called on.

Merged semantics applies, i.e. the result reflects merged polygons rather than individual ones unless raw mode is chosen.

The "mode" argument allows selecting specific edges from polygons. Allowed values are: "convex", "concave", "step", "step_in" and "step_out". "step" generates edges only if they provide a step between two other edges. "step_in" creates edges that make a step towards the inside of the polygon and "step_out" creates edges that make a step towards the outside:

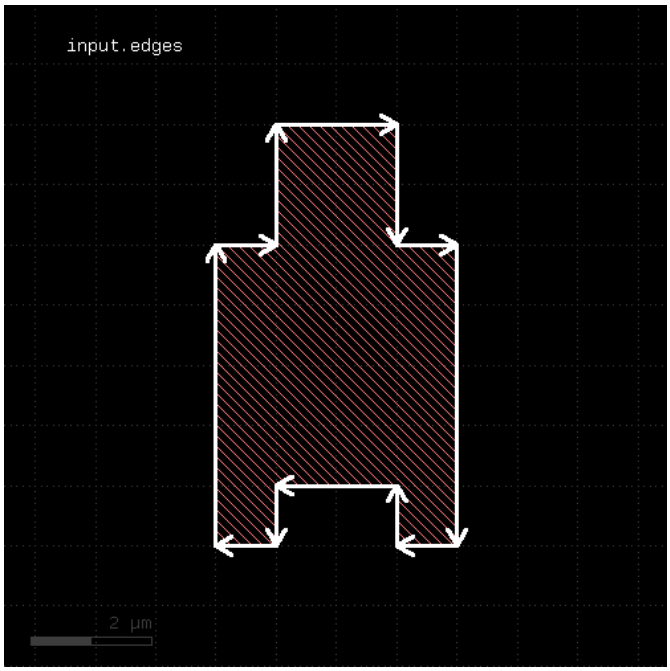
```
out = in.edges(convex)
```

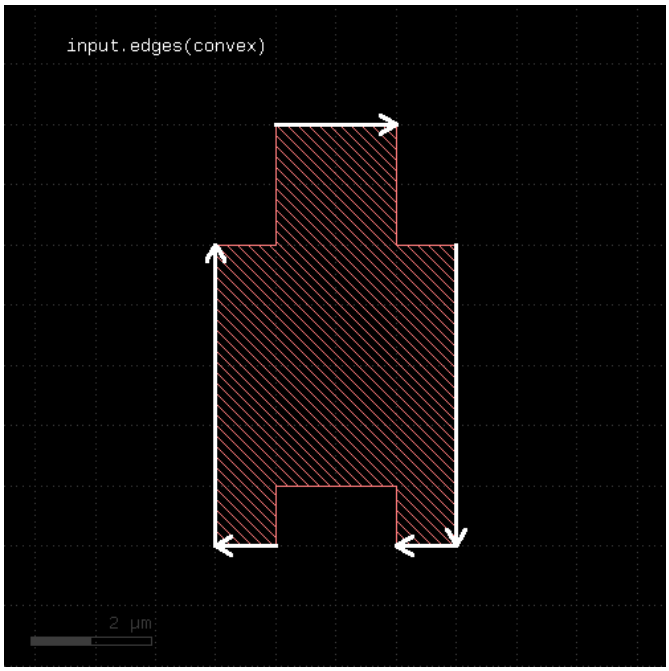
In addition, "not_.." variants are available which selects edges not qualifying for the specific mode:

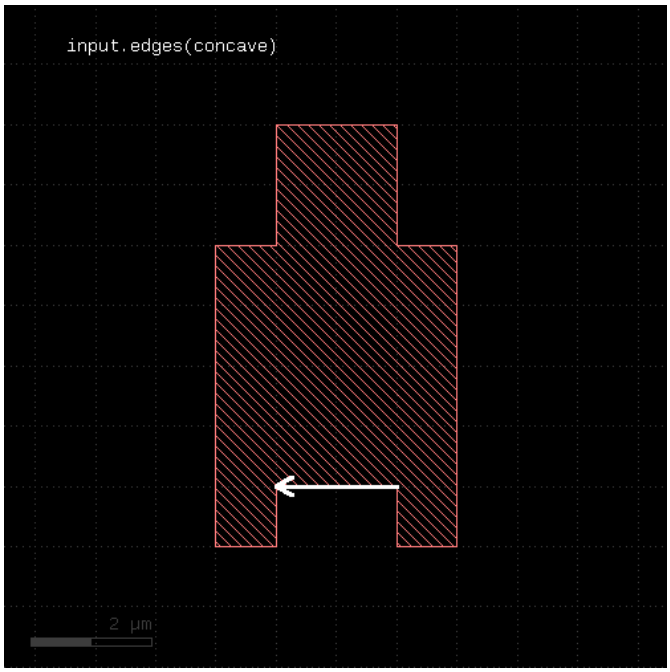
```
out = in.edges(not_convex)
```

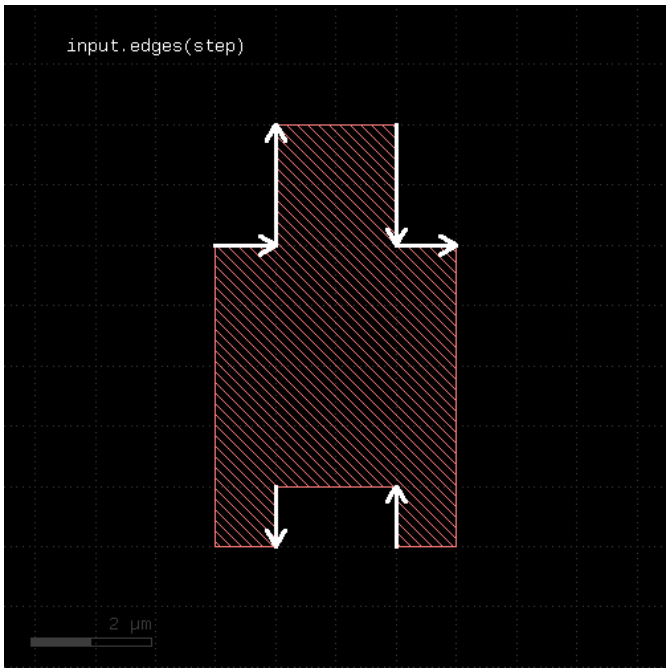
The mode argument is only available for polygon layers.

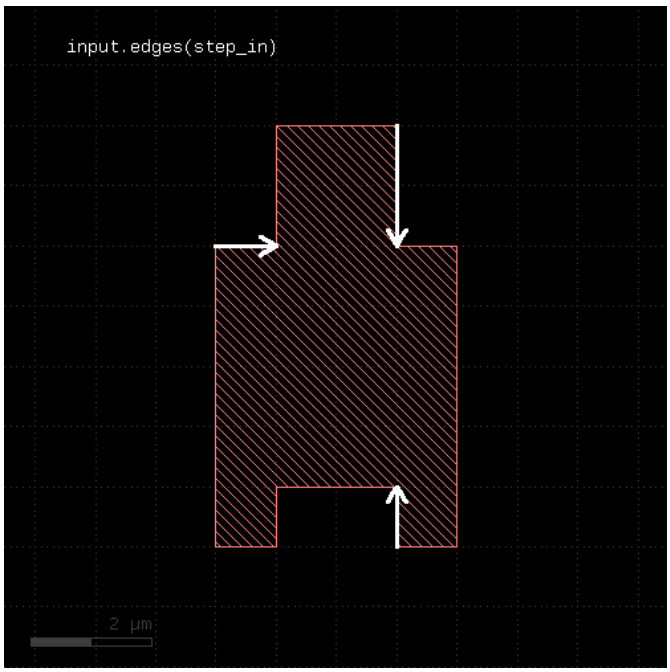
The following images show the effect of the mode argument:

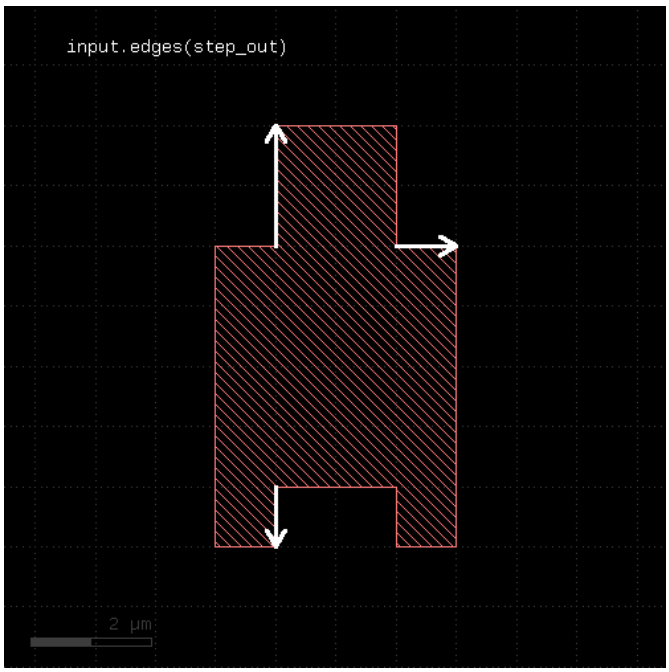












"edges?" - Returns true, if the layer is an edge layer

Usage:

- `layer.edges?`

"enc" - An alias for "enclosing"

Usage:

- `layer.enc(value [, options])`

See enclosing for a description of that method

"enclosed" - An enclosing check (other_layer enclosing layer)

Usage:

- `layer.enclosed(other_layer, value [, options])`

Note: "enclosed" is available as operators for the "universal DRC" function drc within the DRC framework. These variants have more options and are more intuitive to use. See enclosed for more details.

This method checks whether layer is enclosed by (is inside of) other_layer by not less than the given distance value. Locations, where the distance is less will be reported in form of edge pair error markers. Locations, where both edges coincide will be reported as errors as well. Formally such locations form an enclosure with a distance of 0. Locations, where other_layer is inside layer will not be reported as errors. Such regions can be detected by inside or a boolean "not" operation.

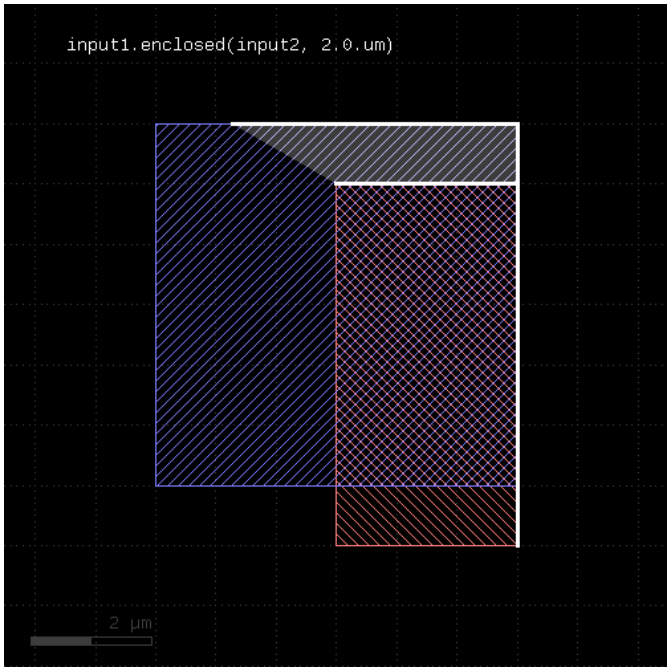
The options are the same as for separation.

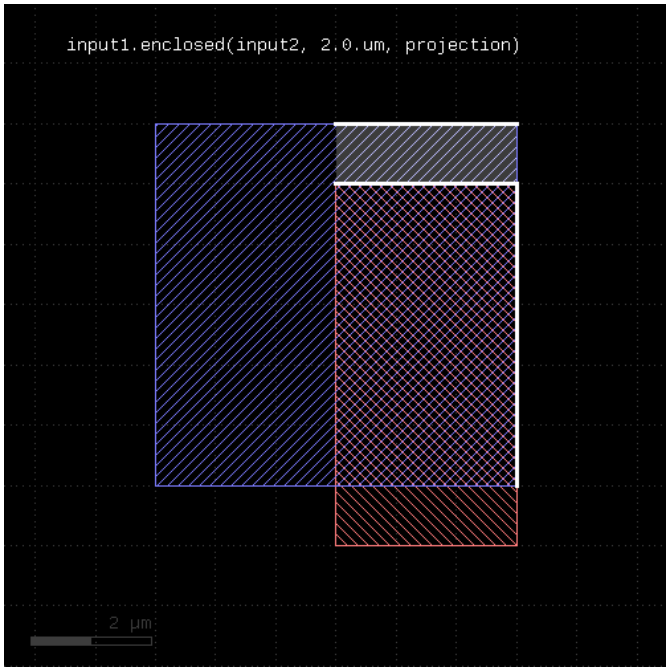
This method is available for edge and polygon layers.

As for the other DRC methods, merged semantics applies.

Distance values can be given as floating-point values (in micron) or integer values (in database units). To explicitly specify the unit, use the unit denominators.

The following images show the effect of two enclosed checks (red: input1, blue: input2):





"enclosing" - An enclosing check (layer enclosing other_layer)

Usage:

- `layer.enclosing(other_layer, value [, options])`
- `layer.enc(other_layer, value [, options])`

Note: "enclosing" and "enc" are available as operators for the "universal DRC" function drc within the DRC framework. These variants have more options and are more intuitive to use. See enclosing for more details.

This method checks whether layer encloses (is bigger than) other_layer by not less than the given distance value. Locations, where the distance is less will be reported in form of edge pair error markers. Locations, where both edges coincide will be reported as errors as well. Formally such locations form an enclosure with a distance of 0. Locations, where other_layer extends outside layer will not be reported as errors. Such regions can be detected by not_inside or a boolean "not" operation.

"enc" is the short form of this method.

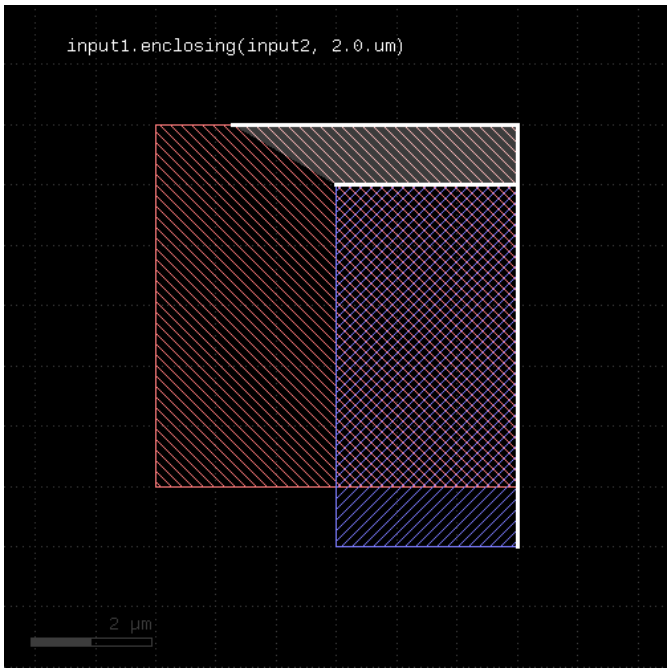
The options are the same as for separation.

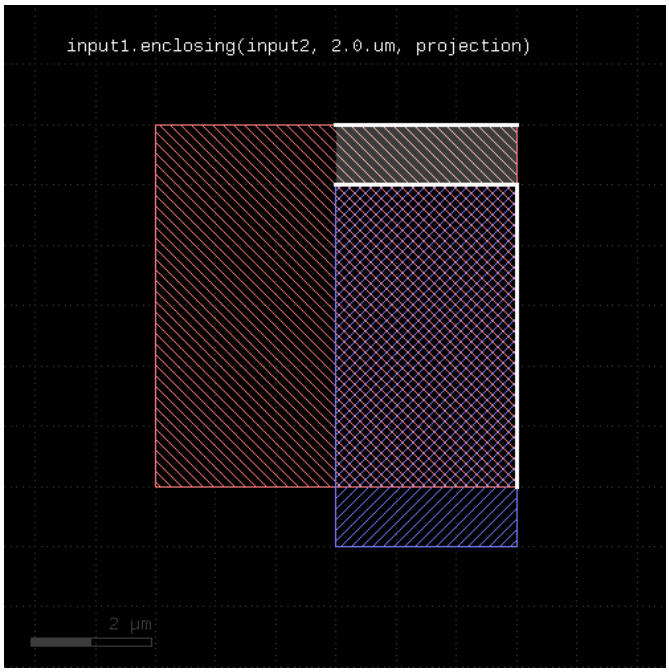
The enclosing method can be applied to both edge or polygon layers. On edge layers the orientation of the edges matters and only edges looking into the same direction are checked.

As for the other DRC methods, merged semantics applies.

Distance values can be given as floating-point values (in micron) or integer values (in database units). To explicitly specify the unit, use the unit denominators.

The following images show the effect of two enclosing checks (red: input1, blue: input2):





"end_segments" - Returns the part at the end of each edge

Usage:

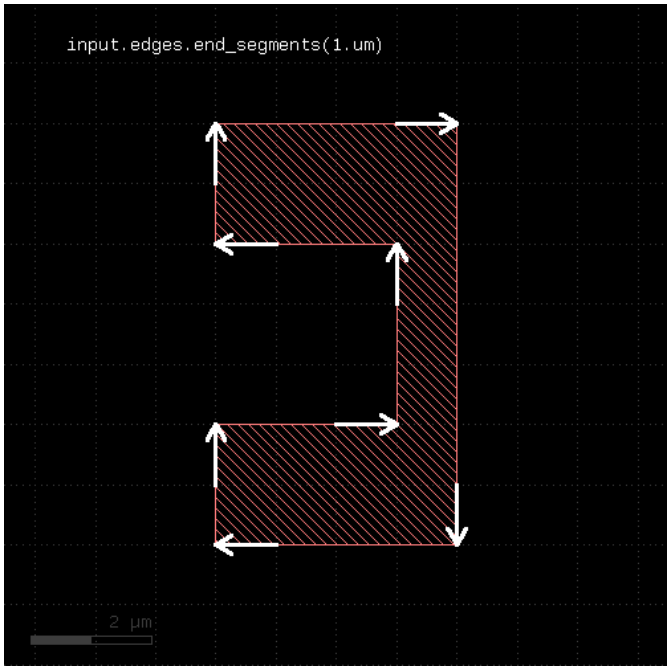
- `layer.end_segments(length)`
- `layer.end_segments(length, fraction)`

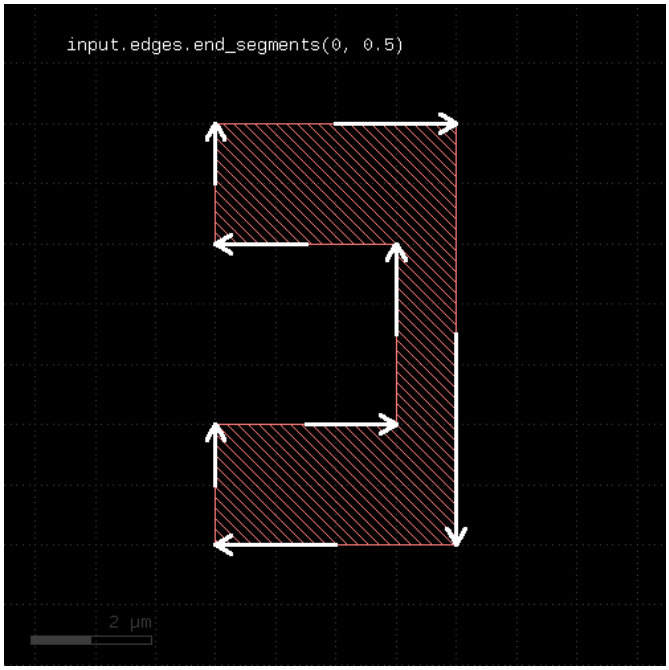
This method will return a partial edge for each edge in the input, located and the end of the original edge. The new edges will share the end point with the original edges, but not necessarily their start point. This method applies to edge layers only. The direction of edges is

defined by the clockwise orientation of a polygon: the end point of the edges will be the terminal point of each edge when walking a polygon in clockwise direction. Or in other words: when looking from start to the end point of an edge, the filled part of the polygon is to the right.

The length of the new edge can be given in two ways: as a fixed length, or a fraction, or both. In the latter case, the length of the resulting edge will be either the fraction or the fixed length, whichever is larger. To specify a length only, omit the fraction argument or leave it at 0. To specify a fraction only, pass 0 to the length argument and specify the fraction in the second parameter. A fraction of 0.5 will result in edges which cover the end half of the edge.

The following images show the effect of the method:





"extended" - Returns polygons describing an area along the edges of the input

Usage:

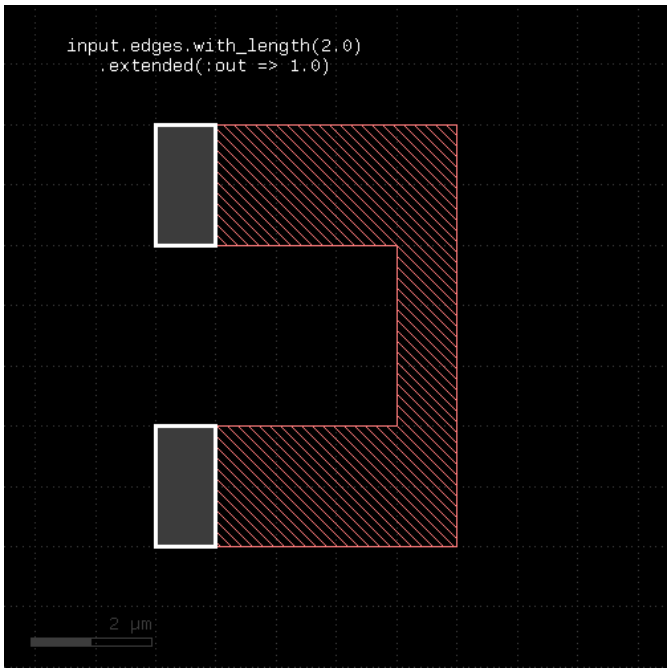
- `layer.extended([:begin => b,] [:end => e,] [:out => o,] [:in => i], [:joined => true])`
- `layer.extended(b, e, o, i)`
- `layer.extended(b, e, o, i, joined)`

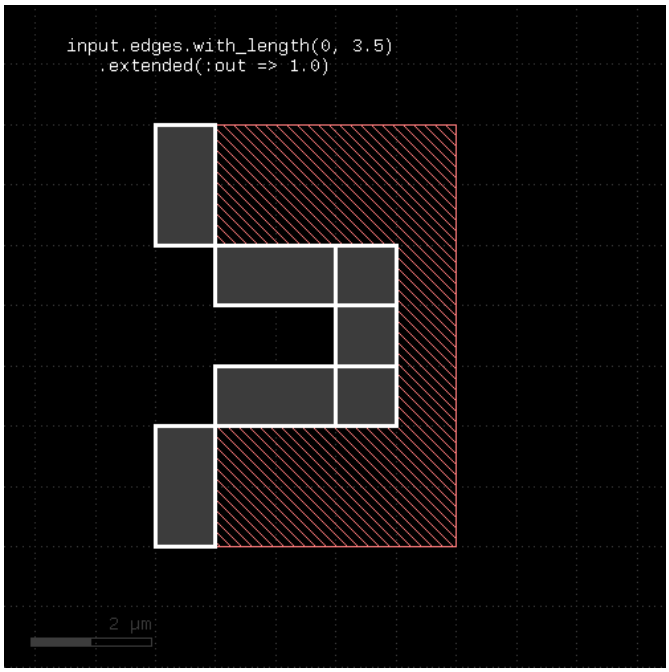
This method is available for edge layers only. It will create a polygon for each edge tracing the edge with certain offsets to the edge. "o" is the offset applied to the outer side of the edge, "i" is the offset applied to the inner side of the edge. "b" is the offset applied at the beginning and "e" is the offset applied at the end.

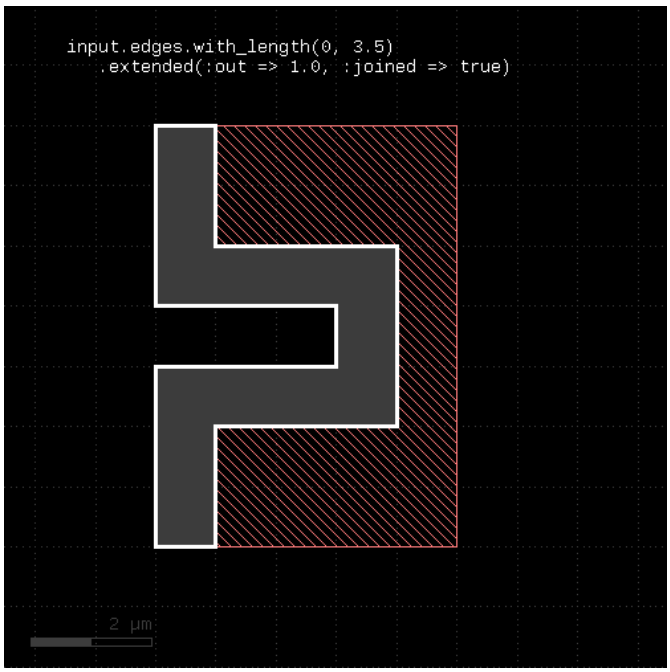
When looking from start to end point, the "inside" side is to the right, while the "outside" side is to the left.

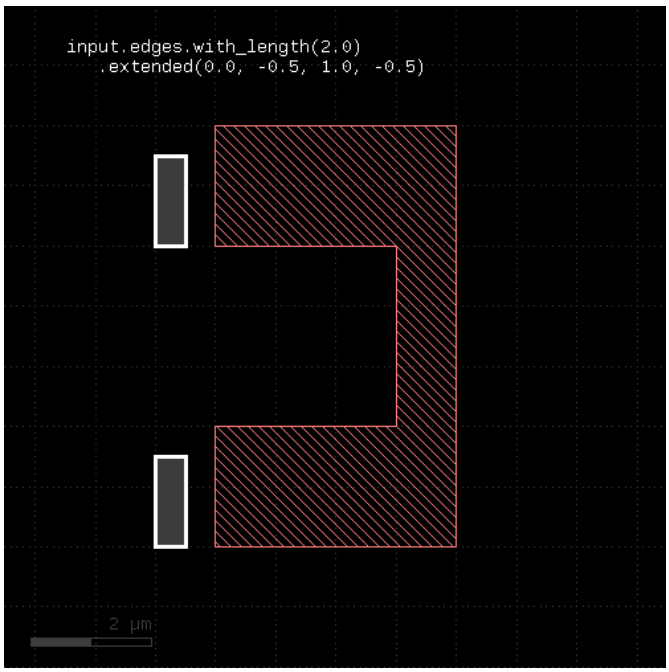
"joined" is a flag, which, if present, will make connected edges behave as a continuous line. Start and end offsets are applied to the first and last unconnected point respectively. Please note that in order to specify joined mode, you'll need to specify "joined" as a keyword in the third form of the method.

The following images show the effects of some parameters:









"extended_in" - Returns polygons describing an area along the edges of the input

Usage:

- `layer.extended_in(d)`

This method applies to edge layers only. Polygons are generated for each edge describing the edge drawn with a certain width extending into the "inside" (the right side when looking from start to end). This method is basically equivalent to the extended method: "extended(0, 0, 0, dist)". A version extending to the outside is extended_out.

"extended_out" - Returns polygons describing an area along the edges of the input

Usage:

- `layer.extended_out(d)`

This method applies to edge layers only. Polygons are generated for each edge describing the edge drawn with a certain width extending into the "outside" (the left side when looking from start to end). This method is basically equivalent to the extended method: "extended(0, 0, dist, 0)". A version extending to the inside is extended_in.

"extent_refs" - Returns partial references to the boundings boxes of the polygons

Usage:

- `layer.extent_refs(fx, fy [, options ])`
- `layer.extent_refs(fx1, fy1, fx2, fx2 [, options ])`
- `layer.extent_refs(ref_spec [, options ])`

This method produces parts of the bounding box of the polygons. It can select either edges, certain points or partial boxes. It can be used the following ways:

- **With a formal specification** : This is an identifier like ":center" or ":left" to indicate which part will be produced.
- **With two floating-point arguments** : These arguments specify a point relative to the bounding box. The first argument is a relative x coordinate where 0.0 means "left side of the bounding box" and 1.0 is the right side. The second argument is a relative y coordinate where 0.0 means "bottom" and 1.0 means "top". The results will be small (2x2 DBU) boxes or point-like edges for edge output
- **With four floating-point arguments** : These arguments specify a box in relative coordinates: a pair of x/y relative coordinate for the first point and another pair for the second point. The results will be boxes or a tilted edge in case of edge output. If the range specifies a finite-area box (height and width are not zero), no adjustment of the boxes will happen for polygon output - i.e. the additional enlargement by 1 DBU which is applied for zero-area boxes does not happen.

The formal specifiers are for points:

- **:center** or **:c** : the center point
- **:bottom_center** or **:bc** : the bottom center point
- **:bottom_left** or **:bl** : the bottom left point
- **:bottom_right** or **:br** : the bottom right point
- **:left** or **:l** : the left point
- **:right** or **:r** : the right point
- **:top_center** or **:tc** : the top center point
- **:top_left** or **:tl** : the top left point
- **:top_right** or **:tr** : the top right point

The formal specifiers for lines are:

- **:bottom** or **:b** : the bottom line

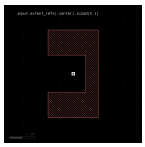
- **:top** or **:t** : the top line
- **:left** or **:l** : the left line
- **:right** or **:r** : the right line

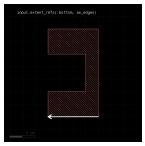
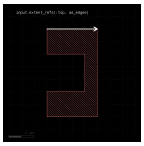
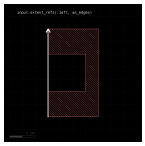
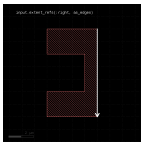
Dots are represented by small (2x2 DBU) boxes or point-like edges with edge output. Lines are represented by narrow or flat (2 DBU) boxes or edges for edge output. Edges will follow the orientation convention for the corresponding edges - i.e. "inside" of the bounding box is on the right side of the edge.

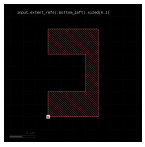
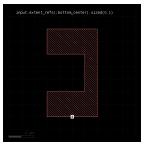
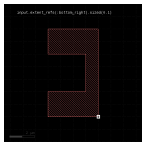
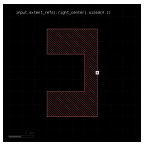
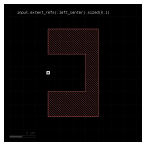
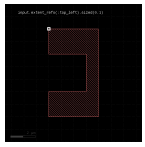
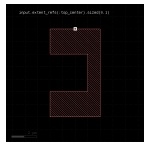
The following additional option controls the output format:

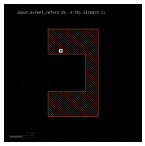
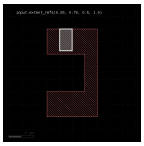
- **as_boxes** : with this option, small boxes will be produced as markers
- **as_dots** or **as_edges** : with this option, point-like edges will be produced for dots and edges will be produced for line-like selections

The following table shows a few applications:



"extents" - Returns the bounding box of each input object

Usage:

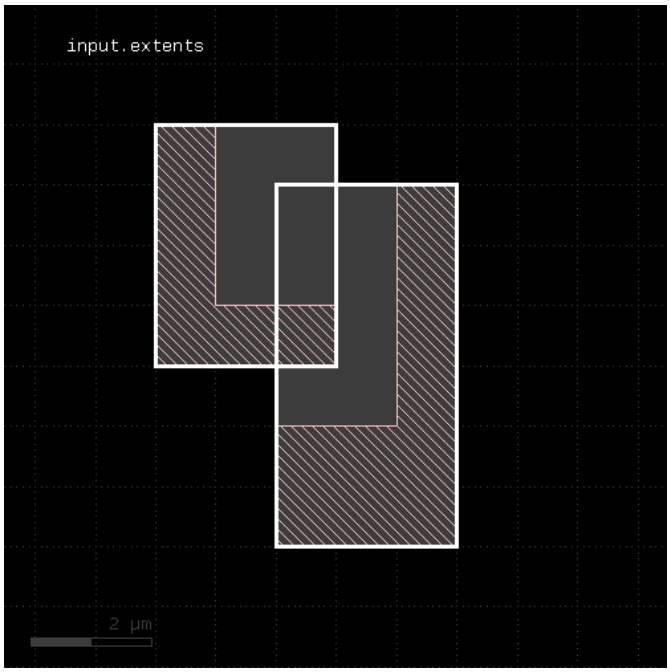
- `layer.extents([ enlargement ])`

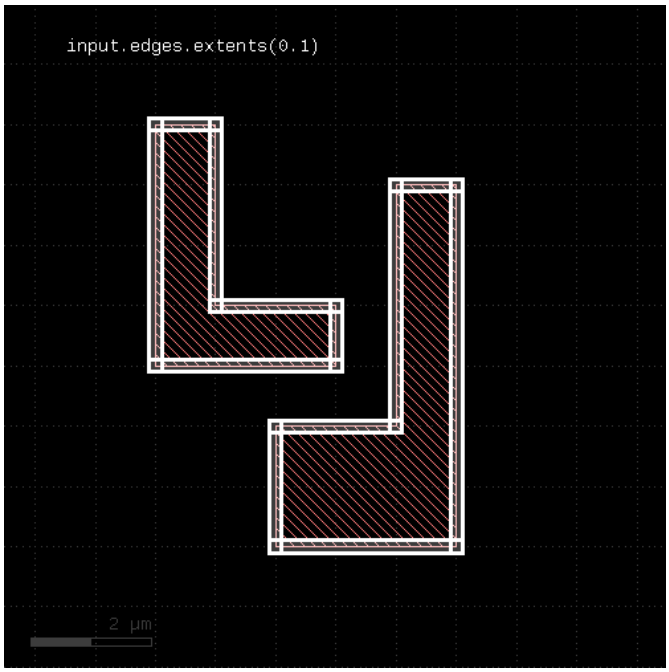
Applies to edge layers, polygon layers on edge pair collections. Returns a polygon layer consisting of boxes for each input object. The boxes enclose the original object.

Merged semantics applies, so the box encloses the merged polygons or edges unless raw mode is chosen (see raw).

The enlargement parameter specifies an optional enlargement which will make zero width/zero height object render valid polygons (i.e. horizontal/vertical edges).

The following images show the effect of the extents method:





"fill" - Fills the region with regular pattern of shapes

Usage:

- `layer.fill([ options ])`

This method will attempt to fill the polygons of the layer with a regular pattern of shapes.

The fill function currently is not available in deep mode.

Options are:

- **hstep(x)** or **hstep(x, y)** : specifies the horizontal step pitch of the pattern. x must be a positive value. A vertical displacement component can be specified too, which results in a skewed pattern.
- **vstep(y)** or **vstep(x, y)** : specifies the vertical step pitch of the pattern. y must be a positive value. A horizontal displacement component can be specified too, which results in a skewed pattern.
- **origin(x, y)** : specifies a fixed point to align the pattern with. This point specifies the location of the reference point for one pattern cell.
- **auto_origin** : lets the algorithm choose the origin. This may result is a slightly better fill coverage as the algorithm is able to determine a pattern origin per island to fill.
- **multi_origin** : lets the algorithm choose the origin and repeats the fill with different origins until no further fill cell can be fitted.
- **fill_pattern(..)** : specifies the fill pattern.

"fill_pattern" generates a fill pattern object. This object is used for configuring the fill pattern content. Fill pattern need to be named. The name will be used for generating the fill cell.

To provide a fill pattern, create a fill pattern object and add shapes to it. The following example creates a fill pattern named "FILL_CELL" and adds a 1x1 micron box on layer 1/0:

```
p = fill_pattern("FILL_CELL")
p.shape(1, 0, box(0.0, 0.0, 1.0, 1.0))
```

See box for details about the box specification. You can also add paths or polygons with path or polygon.

A more compact way of writing this is:

```
p = fill_pattern("FILL_CELL").shape(1, 0, box(0.0, 0.0, 1.0, 1.0))
```

The "shape" method takes several forms:

- `shape(layer, object, object ...)` (1)
- `shape(layer, datatype, object, object ...)` (2)
- `shape(name, object, object ...)` (3)
- `shape(layer_info, object, object ...)` (4)

The first form takes a GDS2 layer number. The datatype is assumed to be 0. The second form takes a GDS layer and datatype number. The third form takes a layer name for layout systems with named layers (like Magic, CIF or DXF). The forth form takes a LayerInfo object to specify the layer. All forms take one to many geometry objects which are written to the respective layer. Geometry objects can either be created using the generator functions (box, polygon, path). The core classes DBox, DPolygon, DPath or DText are also accepted as geometry objects.

The fill pattern can be given a reference point which is used for placing the pattern. The reference point is the one which is aligned with the pattern origin. The following code will assign (-0.5, -0.5) as the reference point for the 1x1 micron rectangle. Hence the reference point is a little below and left of the rectangle which in turn shifts the rectangle fill pattern to the right and up:

```
p = fill_pattern("FILL_CELL")
p.shape(1, 0, box(0.0, 0.0, 1.0, 1.0))
p.origin(-0.5, -0.5)
```

Without a reference point given, the lower left corner of the fill pattern's bounding box will be used as the reference point. The reference point will also defined the footprint of the fill cell - more precisely the lower left corner. When step vectors are given, the fill cell's footprint is taken to be a rectangle having the horizontal and vertical step pitch for width and height respectively. This way the fill cells will be arrange seamlessly. However, the cell's dimensions can be changed, so that the fill cells can overlap or there is a space between the cells. To change the dimensions use the "dim" method.

The following example specifies a fill cell with an active area of -0.5 .. 1.5 in both directions (2 micron width and height). With these dimensions the fill cell's footprint is independent of the step pitch:

```
p = fill_pattern("FILL_CELL")
p.shape(1, 0, box(0.0, 0.0, 1.0, 1.0))
p.origin(-0.5, -0.5)
p.dim(2.0, 2.0)
```

With these ingredients will can use the fill function. The first example fills the polygons of "to_fill" with an orthogonal pattern of 1x1 micron rectangles with a pitch of 2 microns:

```
pattern = fill_pattern("FILL_CELL").shape(1, 0, box(0.0, 0.0, 1.0, 1.0)).origin(-0.5, -0.5)
```

```
to_fill.fill(pattern, hstep(2.0), vstep(2.0))
```

This second example will create a skewed fill pattern in auto-origin mode:

```
pattern = fill_pattern("FILL_CELL").shape(1, 0, box(0.0, 0.0, 1.0, 1.0)).origin(-0.5, -0.5)
to_fill.fill(pattern, hstep(2.0, 1.0), vstep(-1.0, 2.0), auto_origin)
```

The fill function can only work with a target layout for output. It will not work for report output.

The layers generated by the fill cells is only available for input later in the script if the output layout is identical to the input layouts. If you need the area missed by the fill function, try fill_with_left.

"fill_with_left" - Fills the region with regular pattern of shapes

Usage:

- `layer.fill_with_left([ options ])`

This method has the same call syntax and functionality than fill. Other than this method it will return the area not covered by fill cells as a DRC layer.

"first_edges" - Returns the first edges of an edge pair collection

Usage:

- `layer.first_edges`

Applies to edge pair collections only. Returns the first edges of the edge pairs in the collection.

Some checks deliver symmetric edge pairs (e.g. space, width, etc.) for which the edges are commutable. "first_edges" will deliver both edges for such edge pairs.

"flatten" - Flattens the layer

Usage:

- `layer.flatten`

If the layer already is a flat one, this method does nothing. If the layer is a hierarchical layer (an original layer or a derived layer in deep mode), this method will convert it to a flat collection of texts, polygons, edges or edge pairs.

"forget" - Cleans up memory for this layer

Usage:

- `forget`

KLayout's DRC engine is imperative. This means, every command is executed immediately rather than being compiled and executed later. The advantage of this approach is that it allows decisions to be taken depending on the content of a layer and to code functions that operate directly on the layer's content.

However, one drawback is that the engine cannot decide when a layer is no longer required - it may still be used later in the script. So a layer's data is not cleaned up automatically.

In order to save memory for DRC scripts intended for bigger layouts, the DRC script should clean up layers as soon as they are no longer required. The "forget" method will free the memory used for the layer's information.

The recommended approach is:

```
l = ... # compute some layer
...
# once you're done with l:
l.forget
l = nil
```

By setting the layer to nil, it is ensured that it can no longer be accessed.

"hier_count" - Returns the hierarchical number of objects on the layer

Usage:

- `layer.hier_count`

The hier_count is the number of raw objects, not merged regions or edges, with each cell counting once. A high count to hier_count (flat to hierarchical) ratio is an indication of a good hierarchical compression. "hier_count" applies only to original layers without clip regions or cell filters and to layers in deep mode. Otherwise, hier_count gives the same value than count.

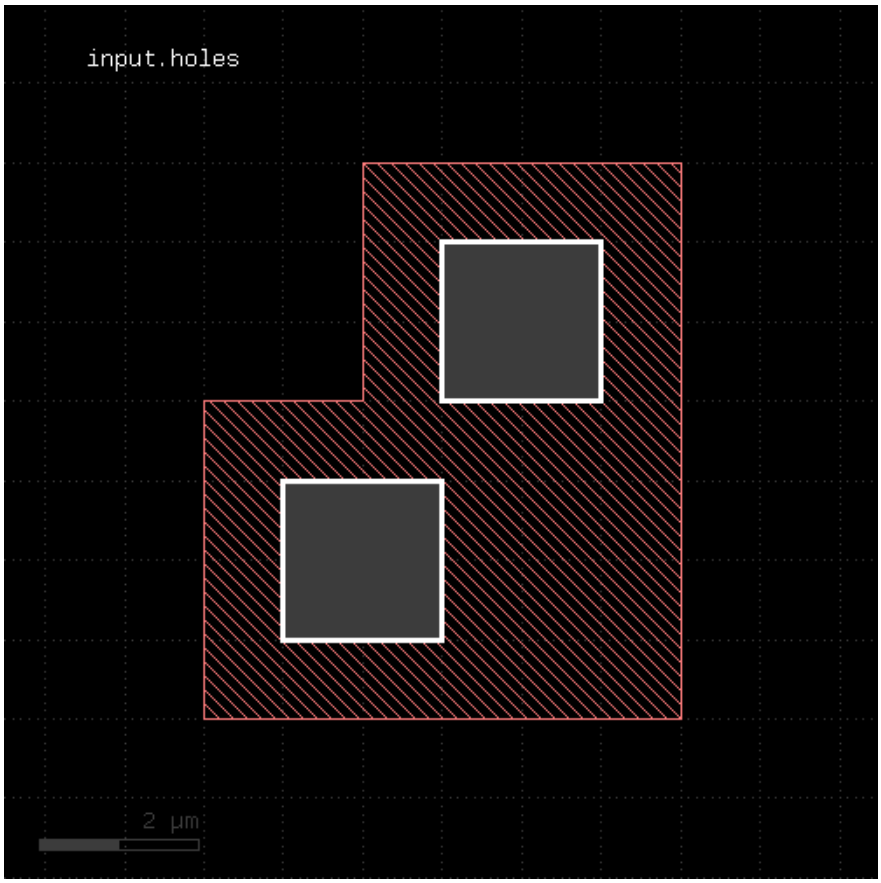
"holes" - Selects all polygon holes from the input

Usage:

- `layer.holes`

This method is available for polygon layers. It will create polygons from all holes inside polygons of the input. Although it is possible, running this method on raw polygon layers will usually not render the expected result, since raw layers do not contain polygons with holes in most cases.

The following image shows the effects of the holes method:



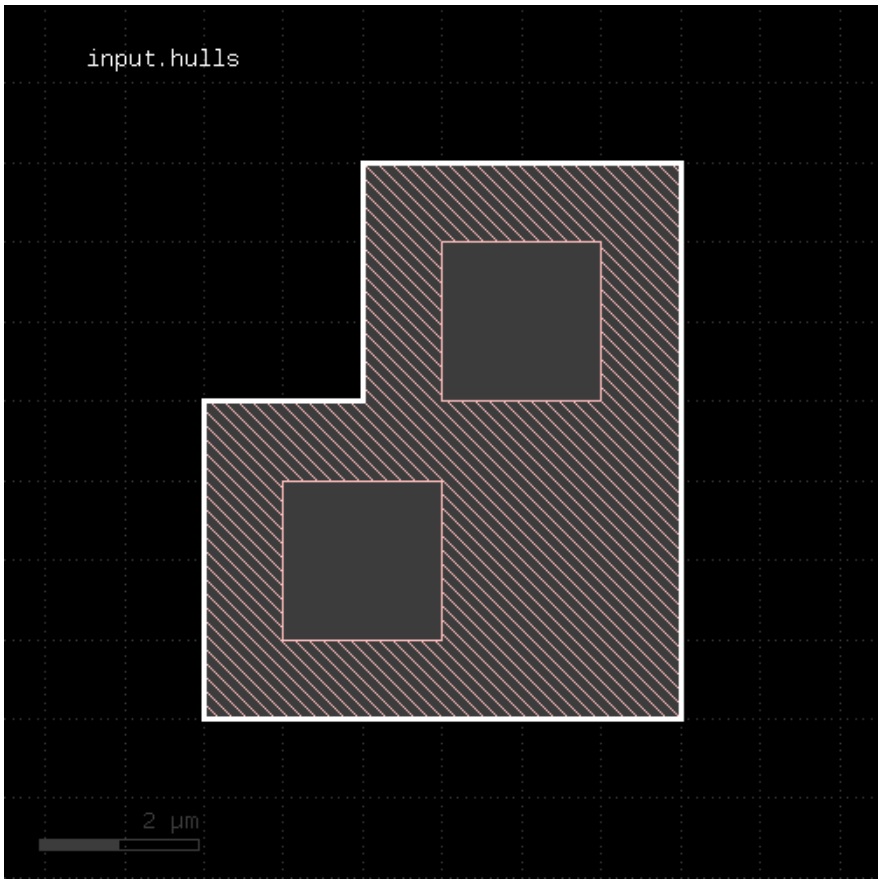
"hulls" - Selects all polygon hulls from the input

Usage:

- `layer.hulls`

This method is available for polygon layers. It will remove all holes from the input and render the hull polygons only. Although it is possible, running this method on raw polygon layers will usually not render the expected result, since raw layers do not contain polygons with holes in most cases.

The following image shows the effects of the hulls method:



"in" - Selects shapes or regions of self which are contained in the other layer

Usage:

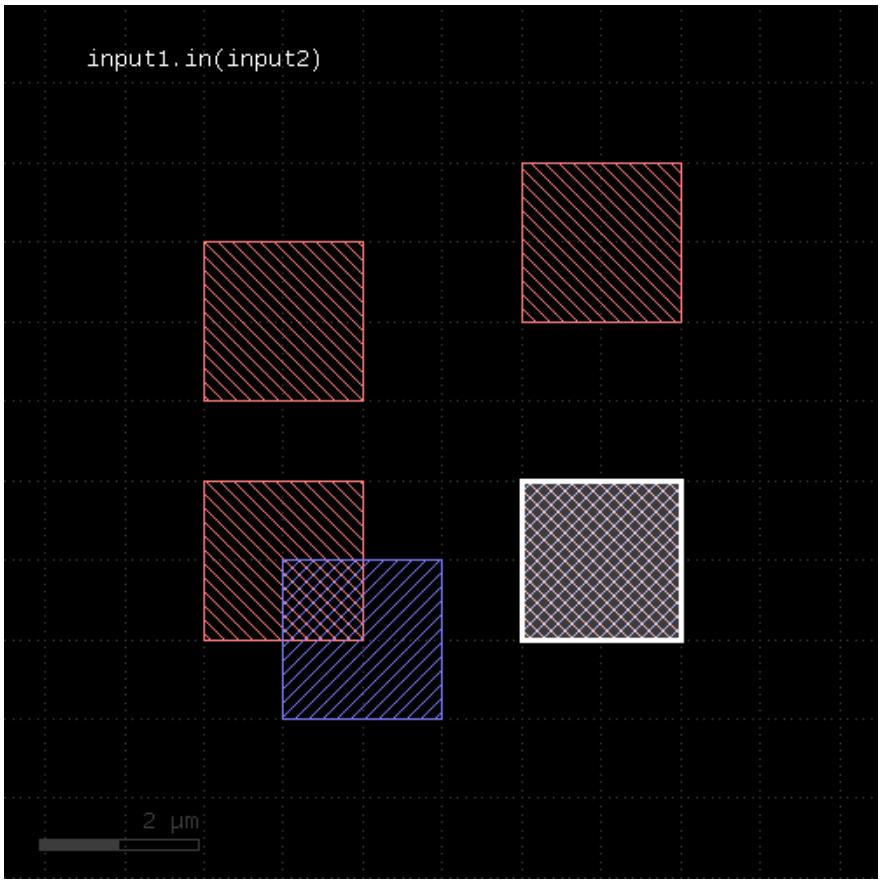
- `layer.in(other)`

This method selects all shapes or regions from self which are contained the other region exactly. It will use individual shapes from self or other if the respective region is in raw mode. If not, it will use coherent regions or combined edges from self or other.

It will return a new layer containing the selected shapes. A method which selects all shapes not contained in the other layer is not_in.

This method is available for polygon and edge layers.

The following image shows the effect of the "in" method (input1: red, input2: blue):



"in_and_out" - Selects shapes or regions of self which are and which are not contained in the other layer

Usage:

- `(in, not_in) = layer.in_and_out(other)`

This method is equivalent to calling in and not_in, but more efficient as it delivers both results in a single call.

"insert" - Inserts one or many objects into the layer

Usage:

- `insert(object, object ...)`

Objects that can be inserted are Edge objects (into edge layers) or DPolygon, DSimplePolygon, Path, DBox (into polygon layers). Convenience methods exist to create such objects (edge, polygon, box and path). However, RBA constructors can used as well.

The insert method is useful in combination with the polygon_layer or edge_layer functions:

```
el = edge_layer
el.insert(edge(0.0, 0.0, 100.0, 0.0)

pl = polygon_layer
pl.insert(box(0.0, 0.0, 100.0, 200.0)
```

"inside" - Selects edges or polygons of self which are inside edges or polygons from the other layer

Usage:

- `layer.inside(other)`

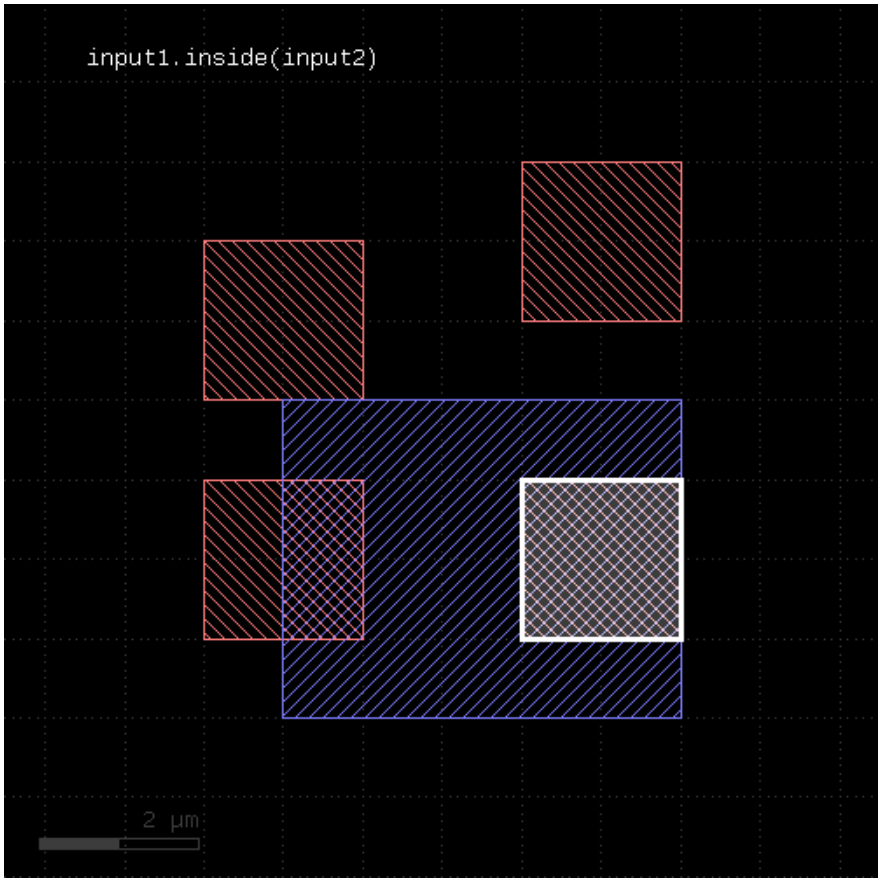
If layer is a polygon layer, the other layer needs to be a polygon layer too. In this case, this method selects all polygons which are completely inside polygons from the other layer.

If layer is an edge layer, the other layer can be polygon or edge layer. In the first case, all edges completely inside the polygons from the other layer are selected. If the other layer is an edge layer, all edges completely contained in edges from the other layer are selected.

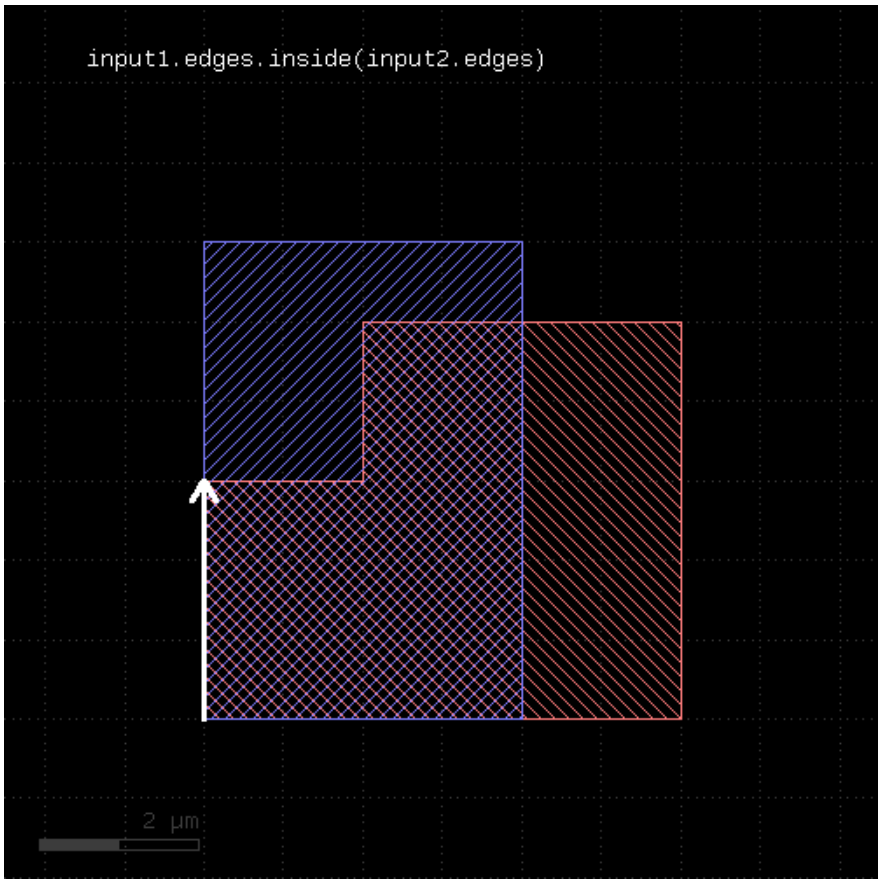
Merged semantics applies - i.e. edges or polygons are joined before the result is computed, unless the layers are in raw mode.

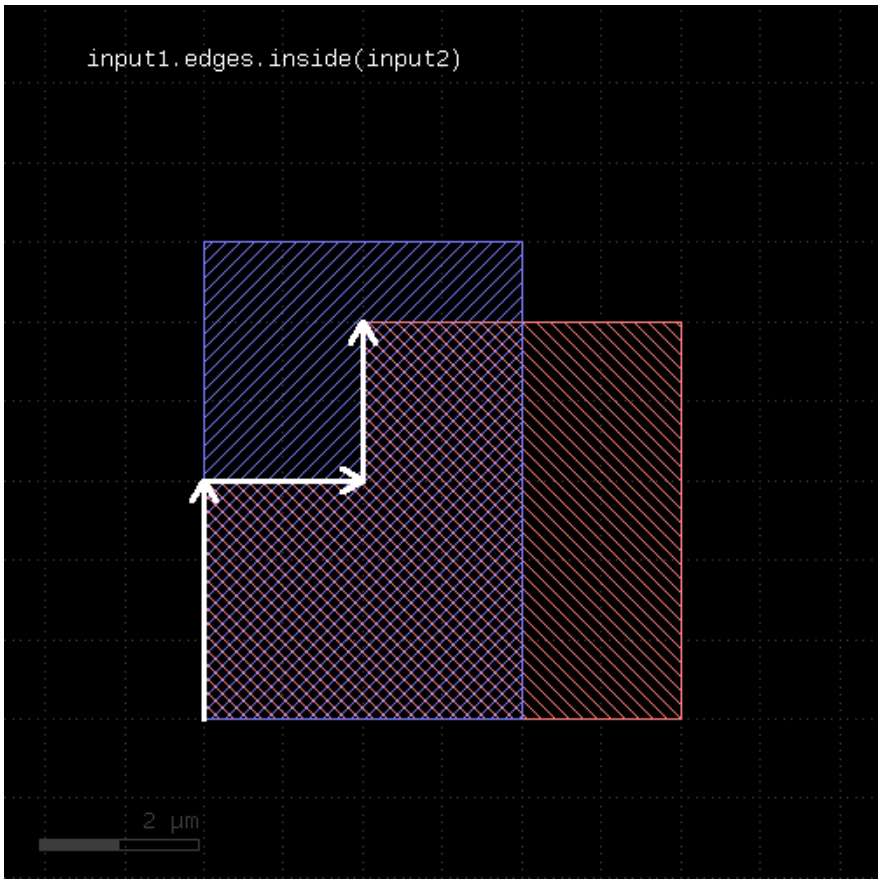
This method returns a new layer containing the selected shapes. A version which modifies self is select_inside. not_inside is a function computing the inverse of inside. split_inside is a function computing both results in a single call. outside is a similar function selecting edges or polygons outside other edges or polygons.

The following image shows the effect of the "inside" method for polygons (input1: red, input2: blue):



The following images show the effect of the "inside" method for edge layers and edge or polygon layers the second input. Note that the edges are computed from the polygons in this example (input1: red, input2: blue):


input1.edges.inside(input2.edges)
2 µm



"inside_outside_part" - Returns the parts of the edges inside and outside the given region

Usage:

- `(inside_part, outside_part) = layer.inside_outside_part(region)`

The method is available for edge layers. The argument must be a polygon layer.

This method returns two layers: the first with the edge parts inside the given region and the second with the parts outside the given region. It is equivalent to calling inside_part and outside_part, but more efficient if both parts need to be computed.

"inside_part" - Returns the parts of the edges inside the given region

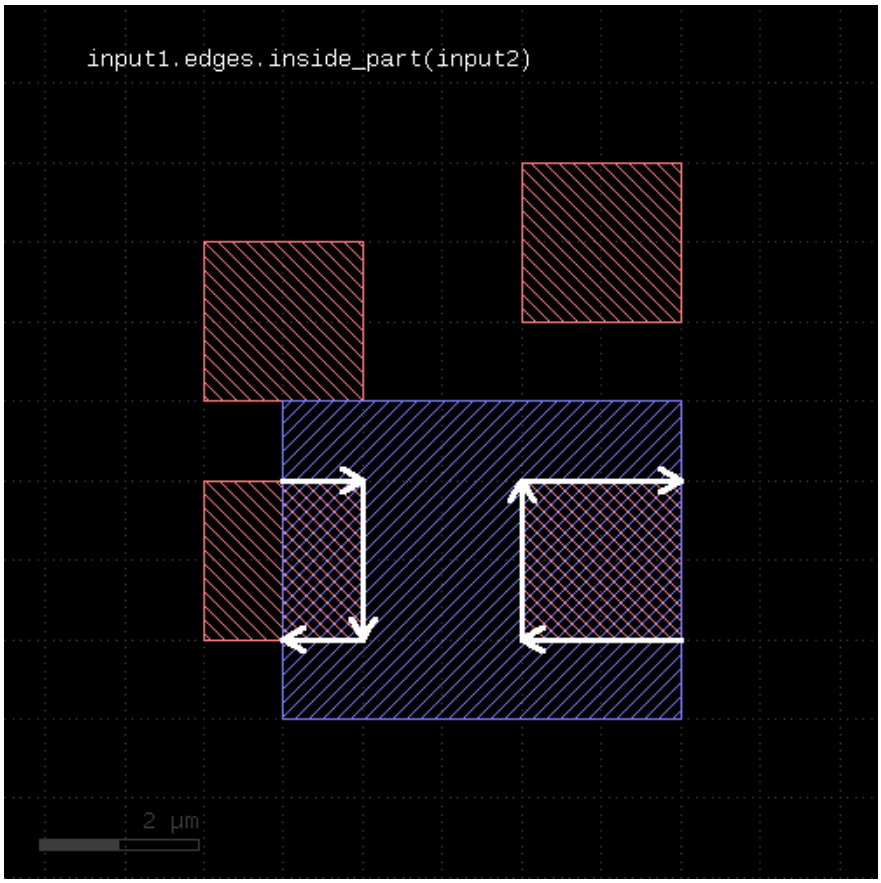
Usage:

- `layer.inside_part(region)`

This method returns the parts of the edges which are inside the given region. This is similar to the "&" operator, but this method does not return edges that are exactly on the boundaries of the polygons of the region.

This method is available for edge layers. The argument must be a polygon layer.

outside_part is a method computing the opposite part. inside_outside_part is a method computing both inside and outside part in a single call.



"interacting" - Selects shapes or regions of self which touch or overlap shapes from the other region

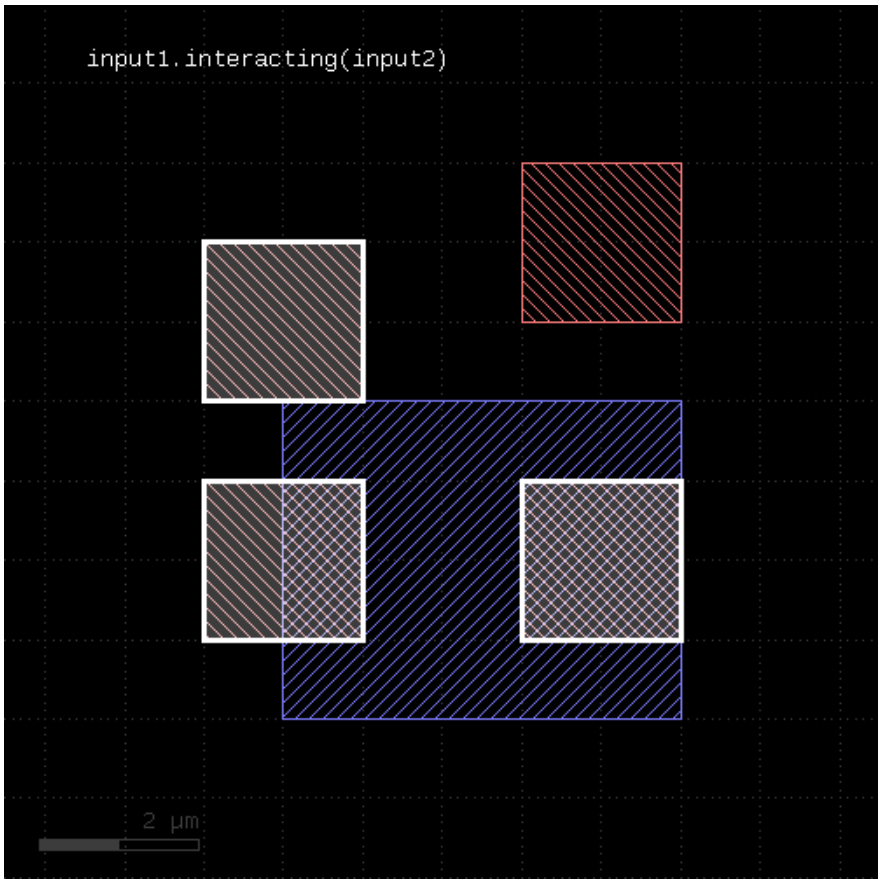
Usage:

- `layer.interacting(other)`
- `layer.interacting(other, min_count)`
- `layer.interacting(other, min_count, max_count)`
- `layer.interacting(other, min_count .. max_count)`

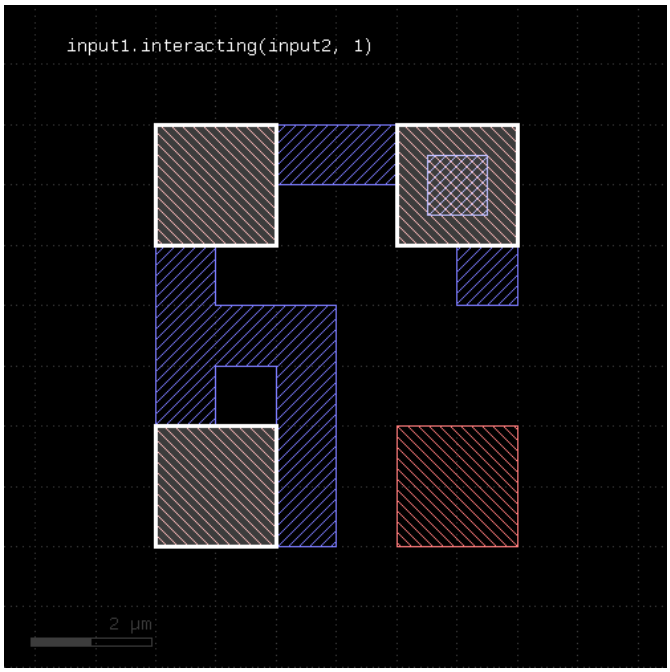
This method selects all shapes or regions from self which touch or overlap shapes from the other region. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. It returns a new layer containing the selected shapes. A version which modifies self is select_interacting.

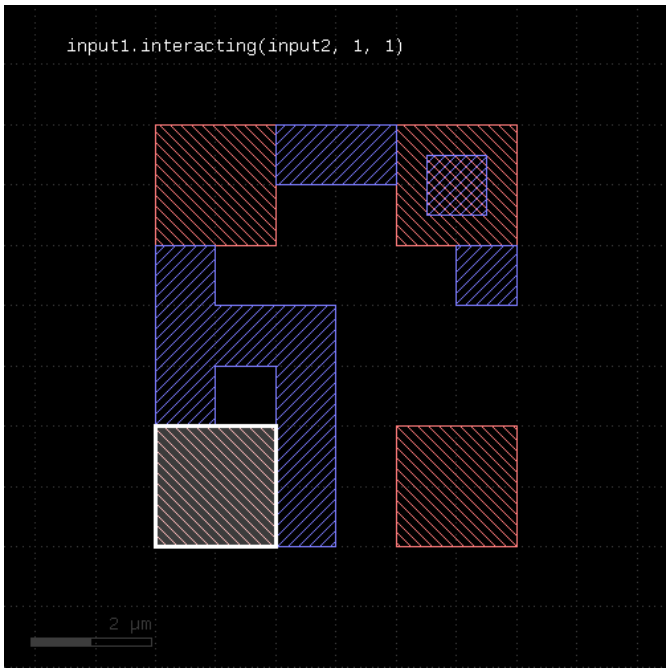
This method is available for polygon, text and edge layers. Edges can be selected with respect to other edges or polygons. Texts can be selected with respect to polygons. Polygons can be selected with respect to edges, texts and other polygons.

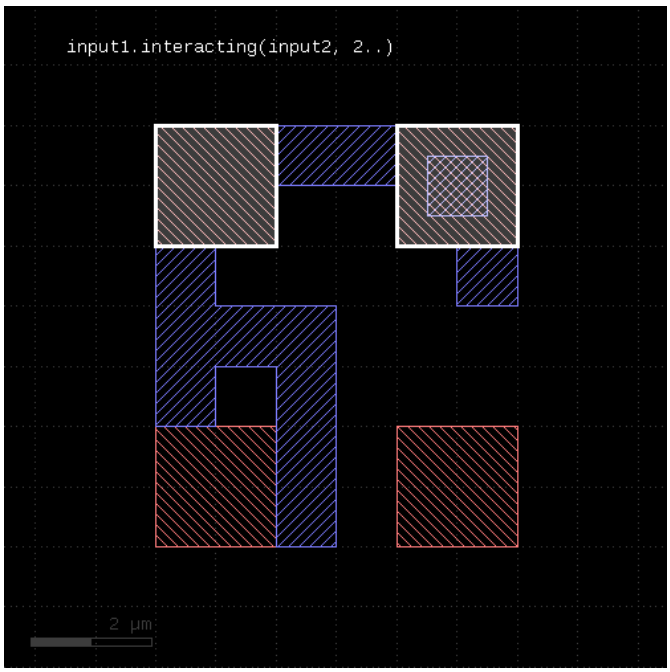
The following image shows the effect of the "interacting" method (input1: red, input2: blue):

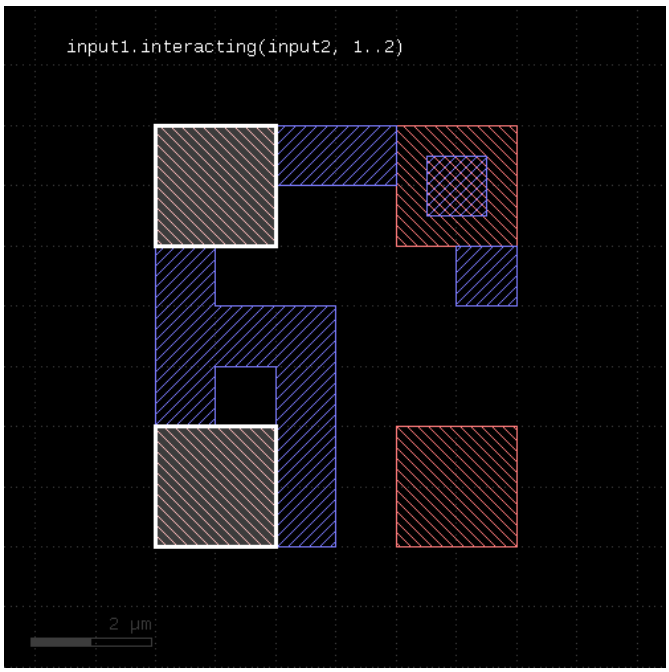


If a single count is given, shapes from self are selected only if they do interact at least with the given number of (different) shapes from the other layer. If a min and max count is given, shapes from self are selected only if they interact with min_count or more, but a maximum of max_count different shapes from the other layer. Two polygons overlapping or touching at two locations are counted as single interactions.









"intersections" - Returns the intersection points of intersecting edge segments for two edge collections

Usage:

- `layer.intersections(edges)`

This operation is similar to the "&" operator, but it does also report intersection points between non-colinear, but intersecting edges. Such points are reported as point-like, degenerated edge objects.

This method is available for edge layers. The argument must be an edge layer.

"is_box?" - Returns true, if the region contains a single box

Usage:

- `layer.is_box?`

The method returns true, if the region consists of a single box only. Merged semantics does not apply - if the region forms a box which is composed of multiple pieces, this method will not return true.

"is_clean?" - Returns true, if the layer is clean state

Usage:

- `layer.is_clean?`

See clean for a discussion of the clean state.

"is_deep?" - Returns true, if the layer is a deep (hierarchical) layer

Usage:

- `layer.is_deep?`

"is_empty?" - Returns true, if the layer is empty

Usage:

- `layer.is_empty?`

"is_merged?" - Returns true, if the polygons of the layer are merged

Usage:

- `layer.is_merged?`

This method will return true, if the polygons of this layer are merged, i.e. they don't overlap and form single continuous polygons. In clean mode, this is ensured implicitly. In raw mode (see raw), merging can be achieved by using the merge method. is_merged? tells, whether calling merge is necessary.

"is_raw?" - Returns true, if the layer is raw state

Usage:

- `layer.is_raw?`

See clean for a discussion of the raw state.

"iso" - An alias for "isolated"

Usage:

- `layer.iso(value [, options])`

See isolated for a description of that method

"isolated" - An inter-polygon isolation check

Usage:

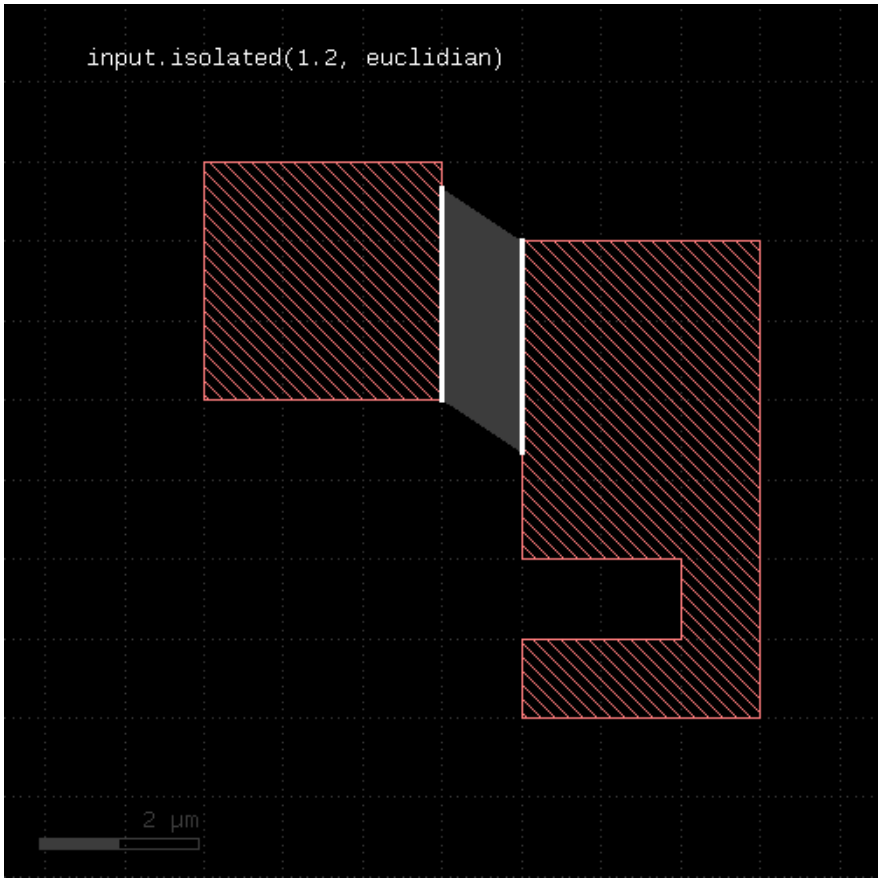
- `layer.isolated(value [, options])`
- `layer.iso(value [, options])`

Note: "isolated" and "iso" are available as operators for the "universal DRC" function Layer#drc within the DRC framework. These variants have more options and are more intuitive to use. See isolated for more details.

See space for a description of this method. "isolated" is the space check variant which checks different polygons only. In contrast to space, the "isolated" method is available for polygon layers only, since only on such layers different polygons can be identified.

"iso" is the short form of this method.

The following image shows the effect of the isolated check:



"join" - Joins the layer with another layer

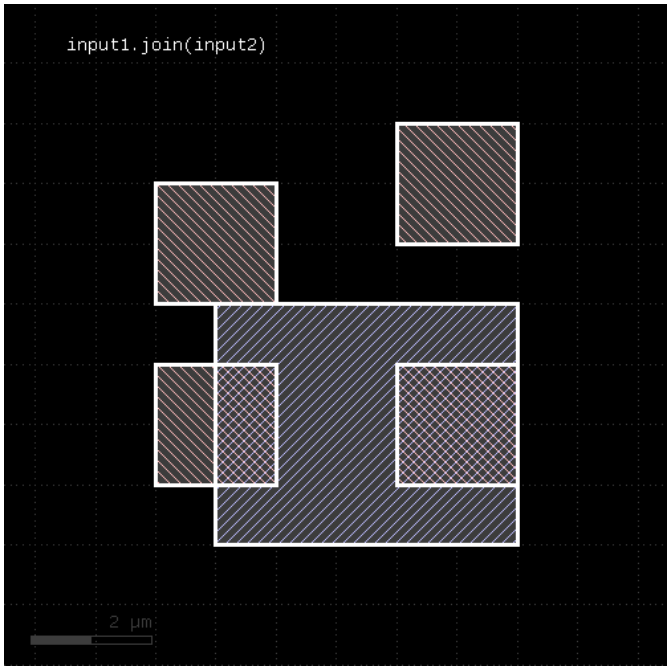
Usage:

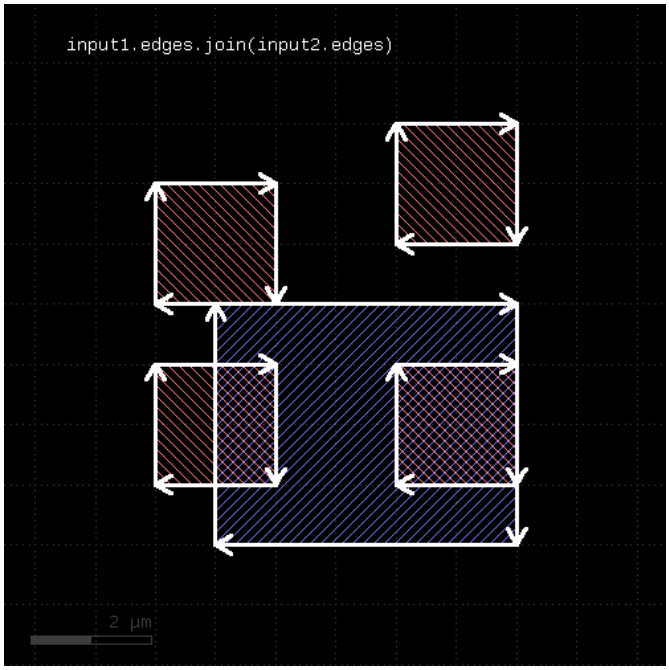
- `layer.join(other)`

The method includes the edges or polygons from the other layer into this layer. It is an alias for the "+" operator.

This method is available for polygon, edge and edge pair layers.

The following images show the effect of the "join" method on polygons and edges (input1: red, input2: blue):





"length" - Returns the total length of the edges in the edge layer

Usage:

- `layer.length`

This method requires an edge layer. It returns the total length of all edges in micron. Merged semantics applies, i.e. before computing the length, the edges are merged unless raw mode is chosen (see raw). Hence in clean mode (see clean), overlapping edges are not counted twice.

"map_props" - Selects properties with certain keys and allows key mapping

Usage:

- `layer.map_props({ key => key_new, .. })`

Similar to select_props, this method will map or filter properties and and take the values from certain keys. In addition, this method allows mapping keys to new keys. Specify a hash argument with old to new keys.

Property values with keys not listed in the hash are removed.

Note that this method returns a new layer with the new properties. The original layer will not be touched.

For example to map key 2 to 1 (integer name keys) and ignore other keys, use:

```
layer1 = input(1, 0, enable_properties)
layer1_mapped = layer1.map_props({ 2 => 1 })
```

See also select_props and remove_props.

"merge" - Merges the layer (modifies the layer)

Usage:

- `layer.merge([overlap_count])`

Like merged, but modifies the input and returns a reference to the new layer.

"merged" - Returns the merged layer

Usage:

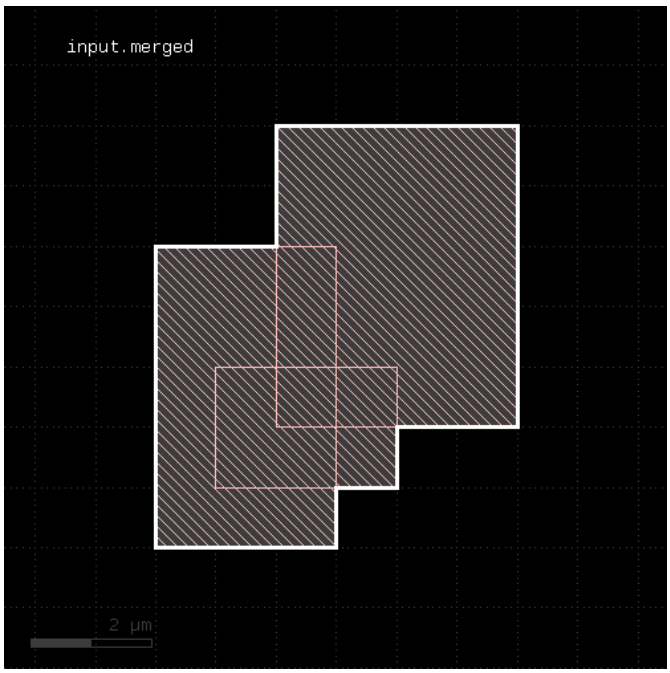
- `layer.merged([overlap_count])`

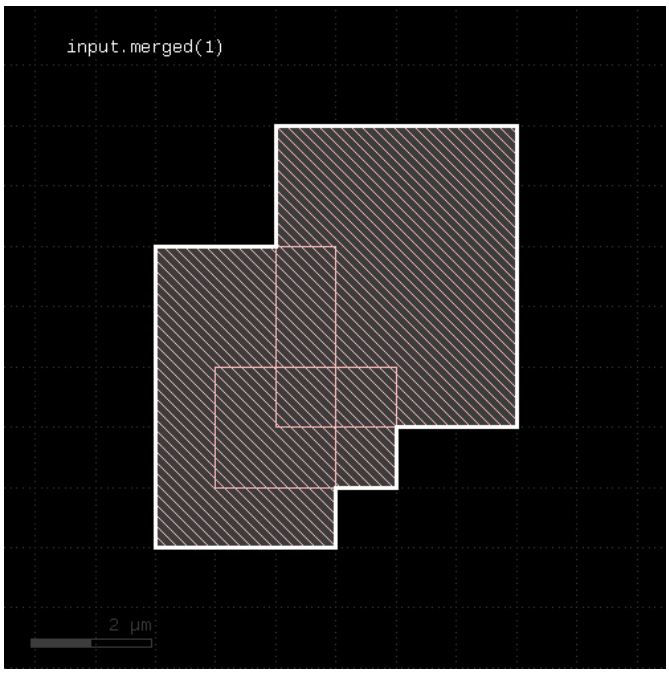
Returns the merged input. Usually, merging is done implicitly using the clean state (which is default). However, in raw state, merging can be enforced by using this method. In addition, this method allows specification of a minimum overlap count, i.e. only where at least the given number of polygons overlap, output is produced. See sized for an application of that.

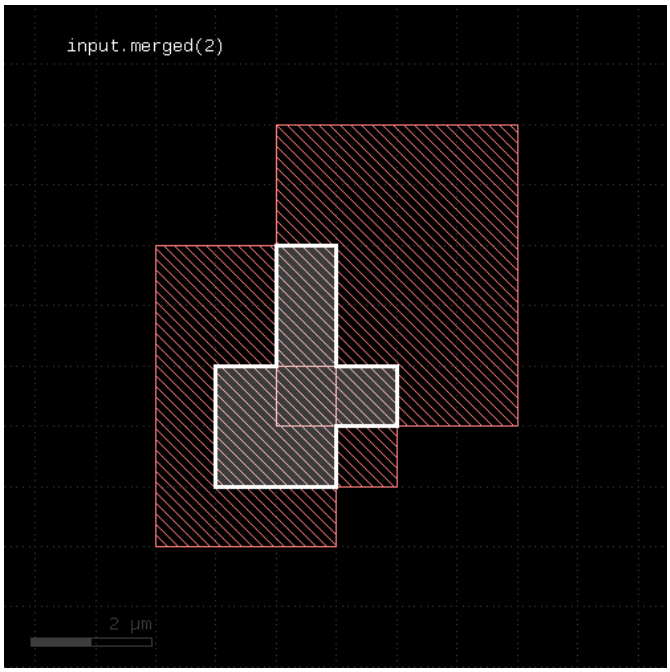
This method works both on edge or polygon layers. Edge merging forms single, continuous edges from coincident and connected individual edges.

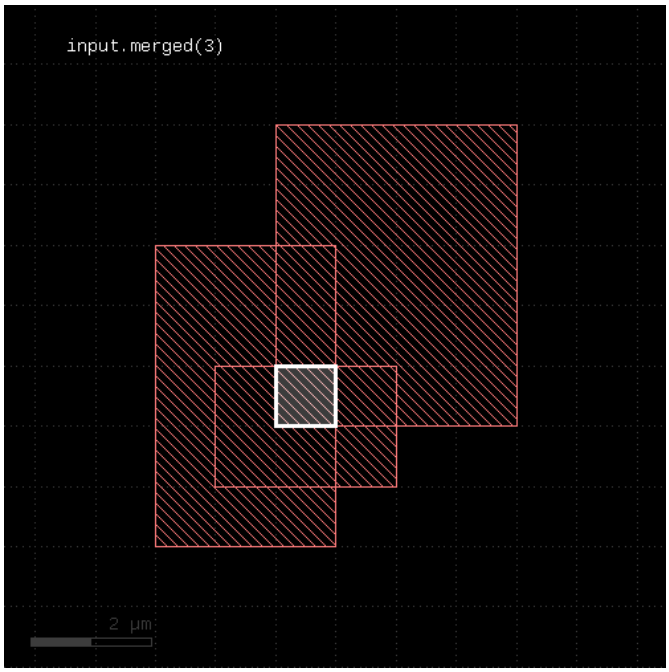
A version that modifies the input layer is merge.

The following images show the effect of various forms of the "merged" method:









"middle" - Returns the center points of the bounding boxes of the polygons

Usage:

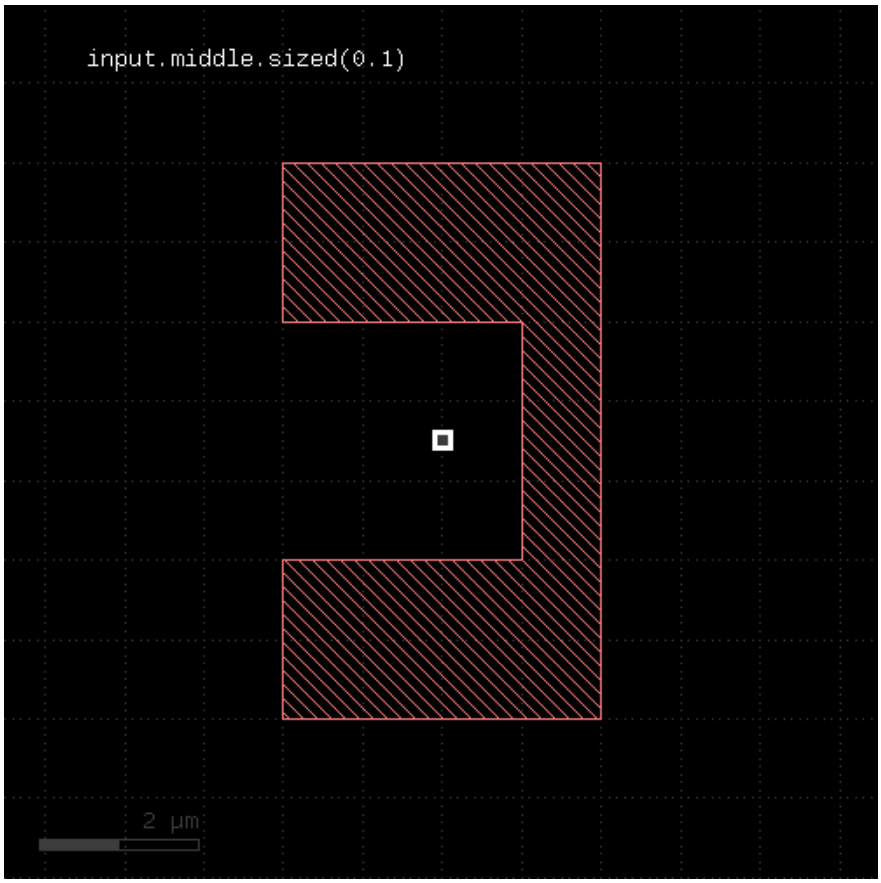
- `layer.middle([ options ])`

This method produces markers on the centers of the polygon's bounding box centers. These markers can be point-like edges or small 2x2 DBU boxes. The latter is the default. A more generic function is extent_refs. "middle" is basically a synonym for "extent_refs(:center)".

The options available are:

- **as_boxes** : with this option, small boxes will be produced as markers
- **as_dots** : with this option, point-like edges will be produced instead of small boxes

The following image shows the effect of this method



"move" - Moves (shifts, translates) a layer (modifies the layer)

Usage:

- `layer.move(dx, dy)`

Moved the input by the given distance. The layer that this method is called upon is modified and the modified version is returned for further processing.

Shift distances can be given as floating-point values (in micron) or integer values (in database units). To explicitly specify the unit, use the unit denominators.

"moved" - Moves (shifts, translates) a layer

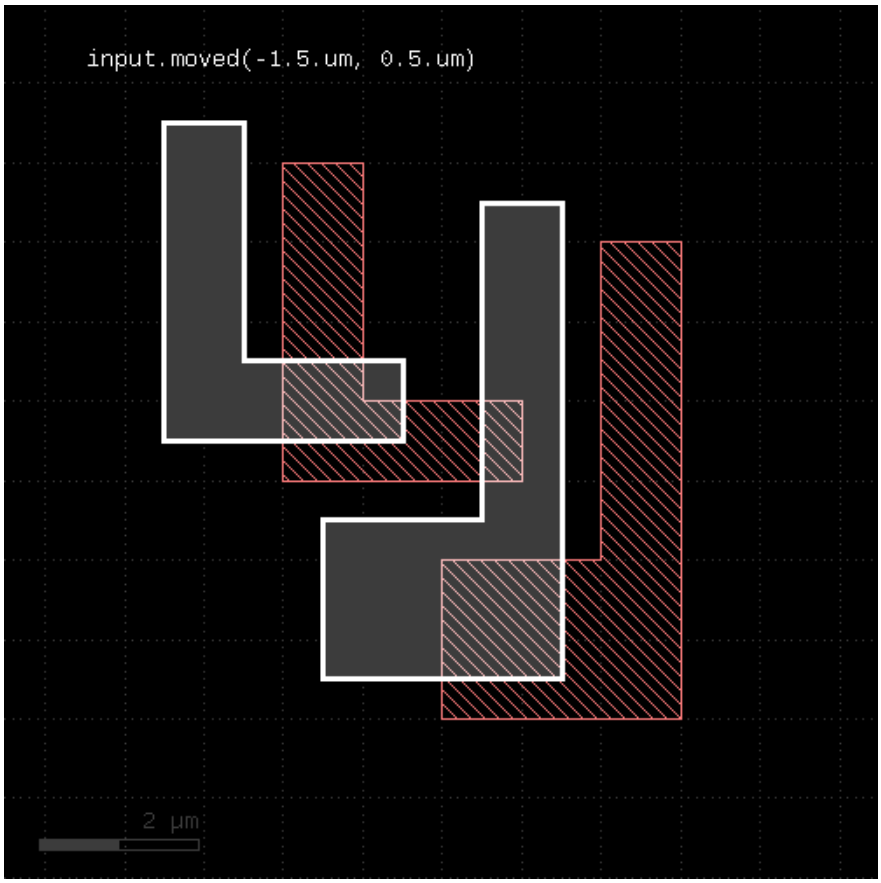
Usage:

- `layer.moved(dx, dy)`

Moves the input layer by the given distance (x, y) and returns the moved layer. The layer that this method is called upon is not modified.

Shift distances can be given as floating-point values (in micron) or integer values (in database units). To explicitly specify the unit, use the unit denominators.

The following images shows the effect of the "moved" method:



"nets" - Pulls net shapes from selected or all nets, optionally annotating nets with properties

Usage:

- `layer.nets`
- `layer.nets(net_filter)`
- `layer.nets(circuit_filter, net_filter)`
- `layer.nets(netter, ...)`
- `layer.nets(prop(key), ...)`
- `layer.nets(prop(key), ...)`

This method needs a layer that has been used in a connect statement. It will take the shapes corresponding to this layer for all or selected nets and attach the net identity in form of a user property.

This way, the resulting shapes can be used in property-constrained boolean operations or DRC checks to implement operations in connected or non-connected mode.

A glob-style name pattern can be supplied to filter nets. Nets are always complete - subnets from subcircuits are not selected. The net name is taken from the net's home circuit (to topmost location where all net connections are formed). You can specify a circuit filter to select nets from certain circuits only or give a Circuit object explicitly.

```
connect(metal1, via1)
```

```
connect(via1, metal2)

metal1_all_nets = metal1.nets
metal1_vdd      = metal1.nets("VDD")
metal1_vdd      = metal1.nets("TOPLEVEL", "VDD")
```

By default, the property key used for the net identity is numerical 0 (integer). You can change the key by giving a property key with the "prop" qualifier. Using "nil" for the key will disable properties:

```
metal1_vdd = metal1.nets("VDD", prop(1))
# disables properties:
metal1_vdd = metal1.nets("VDD", prop(nil))
```

If a custom netter object has been used for the construction of the connectivity, pass it to the "nets" method among the other arguments.

"non_rectangles" - Selects all polygons from the input which are not rectangles

Usage:

- `layer.non_rectangles`

This method is available for polygon layers. By default "merged" semantics applies, i.e. all polygons are merged before non-rectangles are selected (see clean and raw).

"non_rectilinear" - Selects all non-rectilinear polygons from the input

Usage:

- `layer.non_rectilinear`

This method is available for polygon layers. By default "merged" semantics applies, i.e. all polygons are merged before non-rectilinear polygons are selected (see clean and raw).

"non_squares" - Selects all polygons from the input which are not squares

Usage:

- `layer.non_rectangles`

This method is available for polygon layers. By default "merged" semantics applies, i.e. all polygons are merged before non-squares are selected (see clean and raw).

"non_strict" - Marks a layer for non-strict handling

Usage:

- `layer.non_strict`

See strict for details about this option.

This feature has been introduced in version 0.23.2.

"not" - Boolean NOT operation

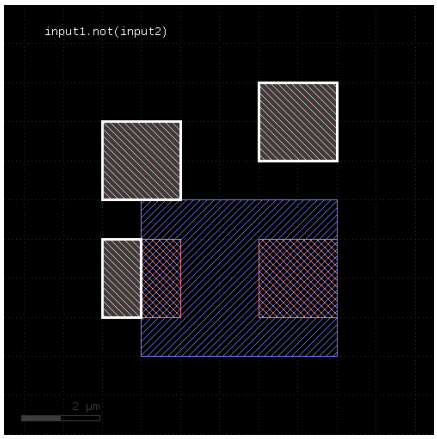
Usage:

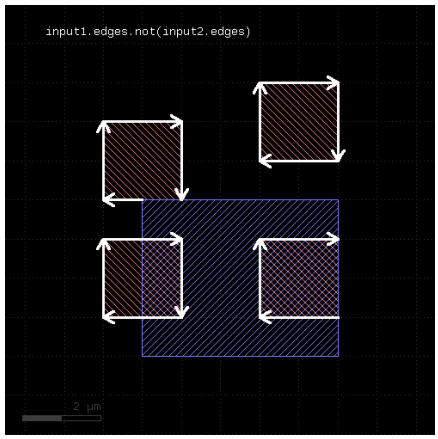
- `layer.not(other [, prop_constraint ])`

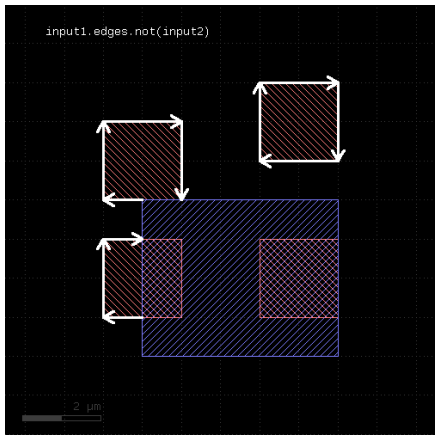
The method computes a boolean NOT between self and other. It is an alias for the "-" operator.

This method is available for polygon and edge layers. If the first operand is an edge layer and the second is an edge layer, the result will be the edges of the first operand which are outside the polygons of the second operand.

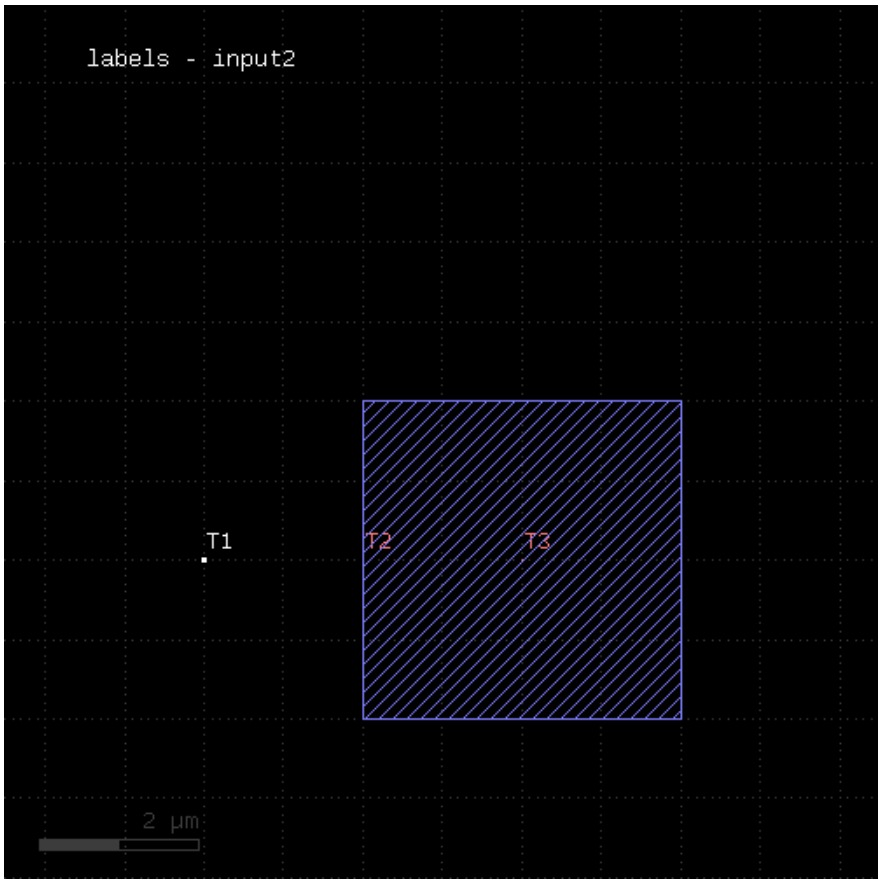
The following images show the effect of the "not" method on polygons and edges (input1: red, input2: blue):







The NOT operation can be applied between a text and a polygon layer. In this case, the texts outside the polygons will be written to the output (labels: red, input2: blue):



When a properties constraint is given, the operation is performed only between shapes with the given relation. Together with the ability to provide net-annotated shapes through the nets method, this allows constraining the boolean operation to shapes from the same or from different nets.

See prop_eq, prop_ne and prop_copy for details.

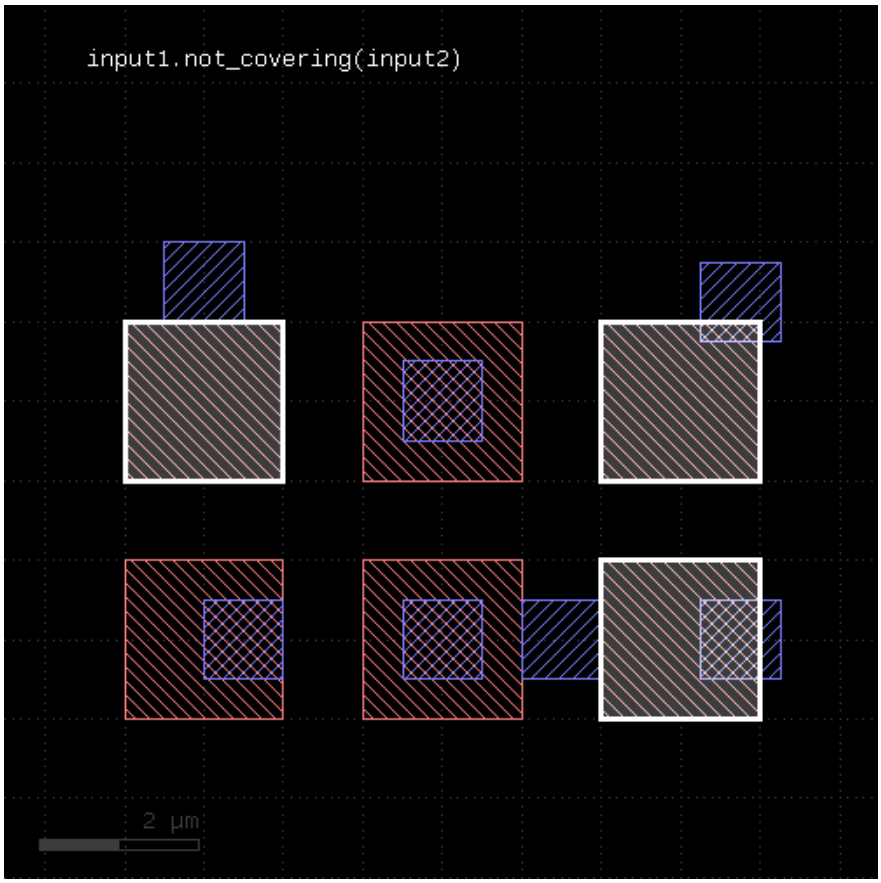
"not_covering" - Selects shapes or regions of self which do not cover (enclose) one or more shapes from the other region

Usage:

- `layer.not_covering(other)`
- `layer.not_covering(other, min_count)`
- `layer.not_covering(other, min_count, max_count)`
- `layer.not_covering(other, min_count .. max_count)`

This method selects all shapes or regions from self which do not cover shapes from the other region. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. This method returns the inverse of covering and provides the same options.

The following image shows the effect of the "not_covering" method:



This method is available for polygons only. It returns a new layer containing the selected shapes. A version which modifies self is select_not_covering.

"not_in" - Selects shapes or regions of self which are not contained in the other layer

Usage:

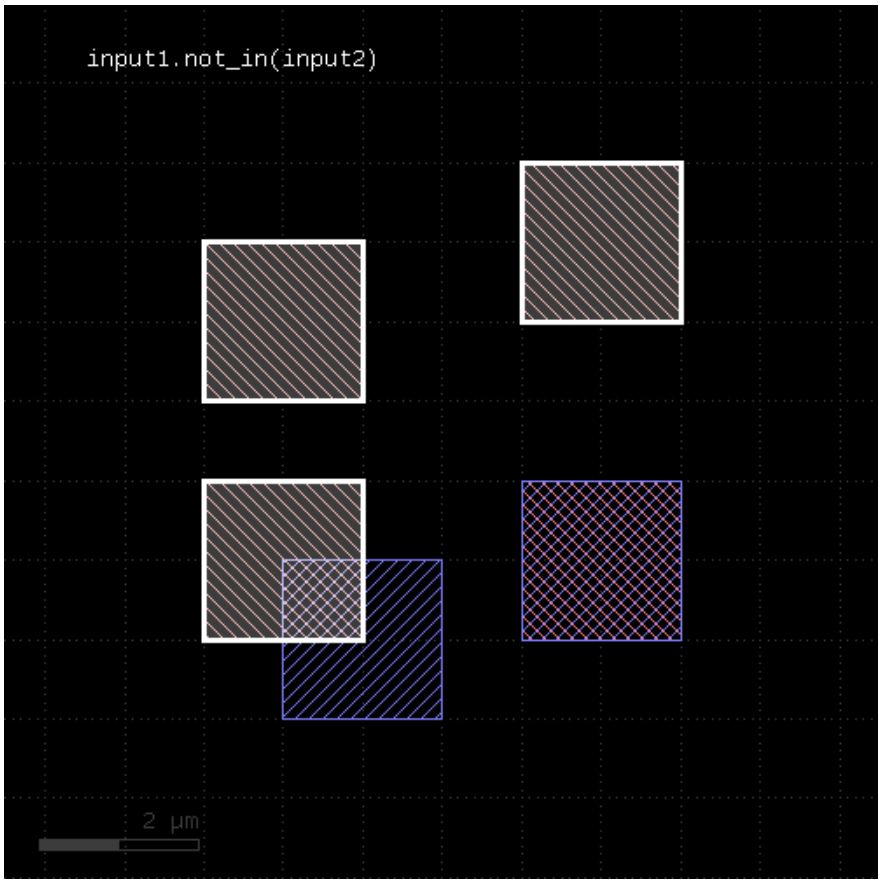
- `layer.not_in(other)`

This method selects all shapes or regions from self which are not contained the other region exactly. It will use individual shapes from self or other if the respective region is in raw mode. If not, it will use coherent regions or combined edges from self or other.

It will return a new layer containing the selected shapes. A method which selects all shapes contained in the other layer is in.

This method is available for polygon and edge layers.

The following image shows the effect of the "not_in" method (input1: red, input2: blue):



"not_inside" - Selects edges or polygons of self which are not inside edges or polygons from the other layer

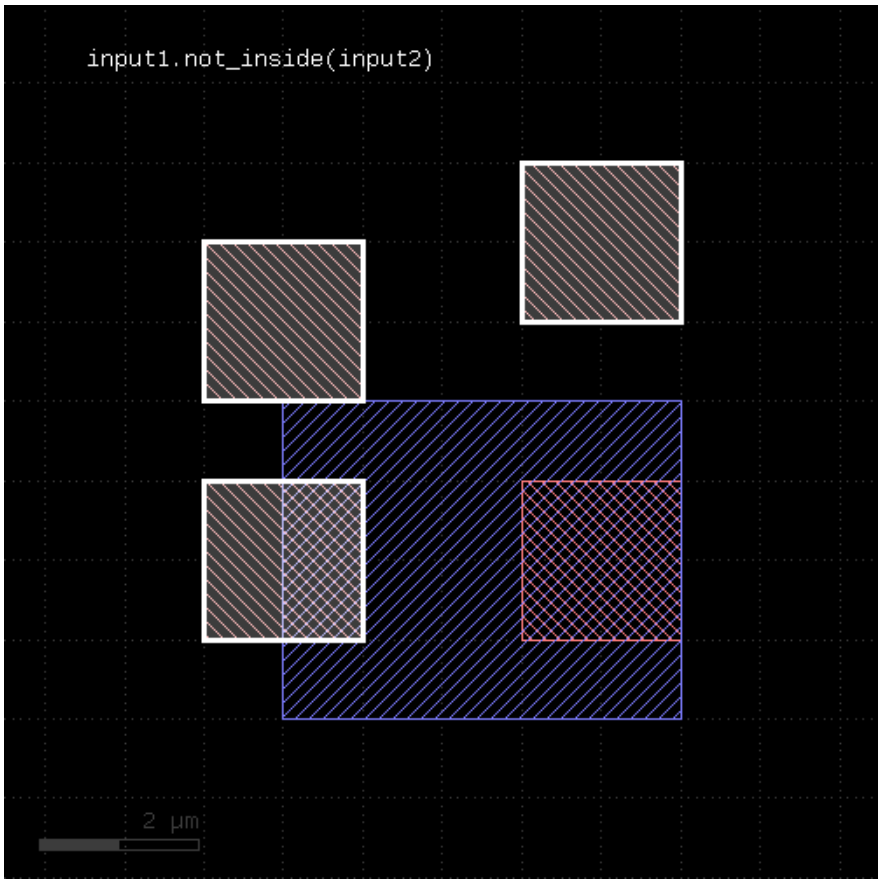
Usage:

- `layer.not_inside(other)`

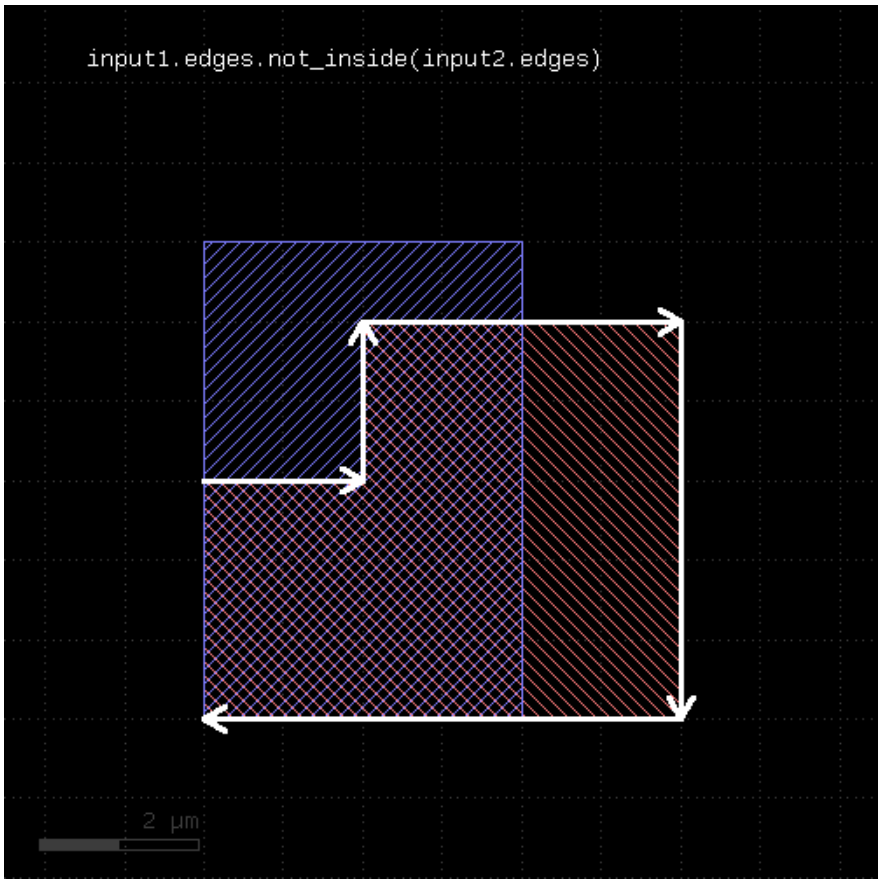
This method computes the inverse of inside - i.e. edges or polygons from the layer not being inside polygons or edges from the other layer.

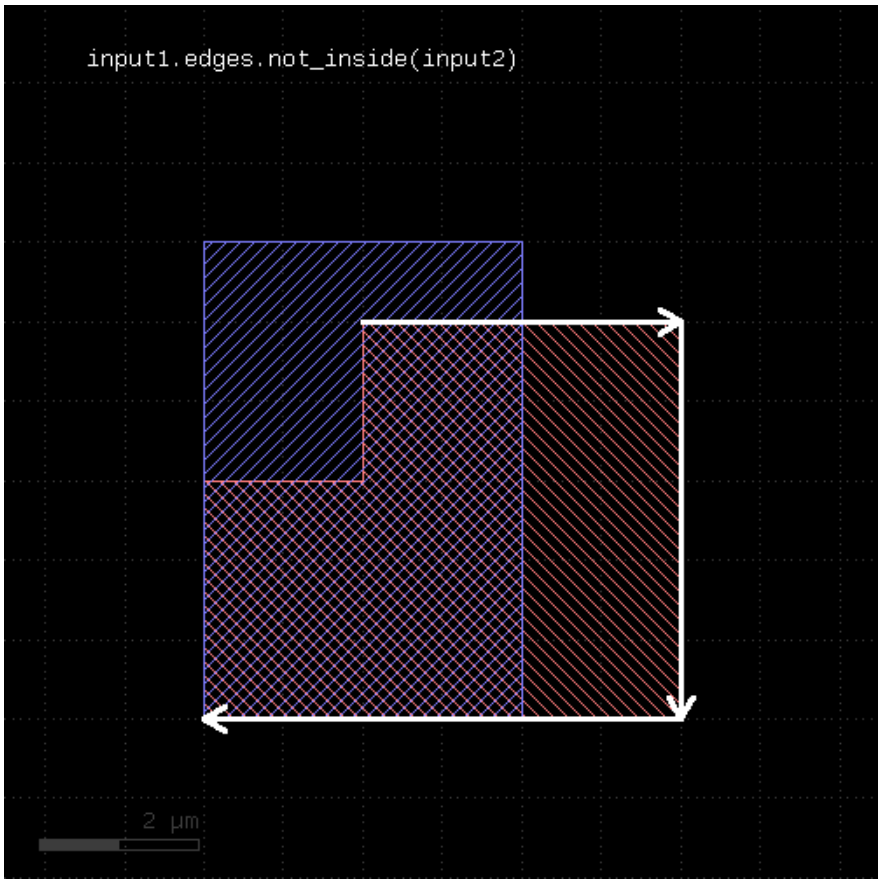
This method returns a new layer containing the selected shapes. A version which modifies self is select_not_inside. split_inside is a function computing both results of inside and not_inside in a single call. outside is a similar function selecting edges or polygons outside other edges or polygons. Note that "outside" is not the same than "not inside".

The following image shows the effect of the "not_inside" method for polygon layers (input1: red, input2: blue):



The following images show the effect of the "not_inside" method for edge layers and edge or polygon layers the second input. Note that the edges are computed from the polygons in this example (input1: red, input2: blue):


input1.edges.not_inside(input2.edges)
2 µm



"not_interacting" - Selects shapes or regions of self which do not touch or overlap shapes from the other region

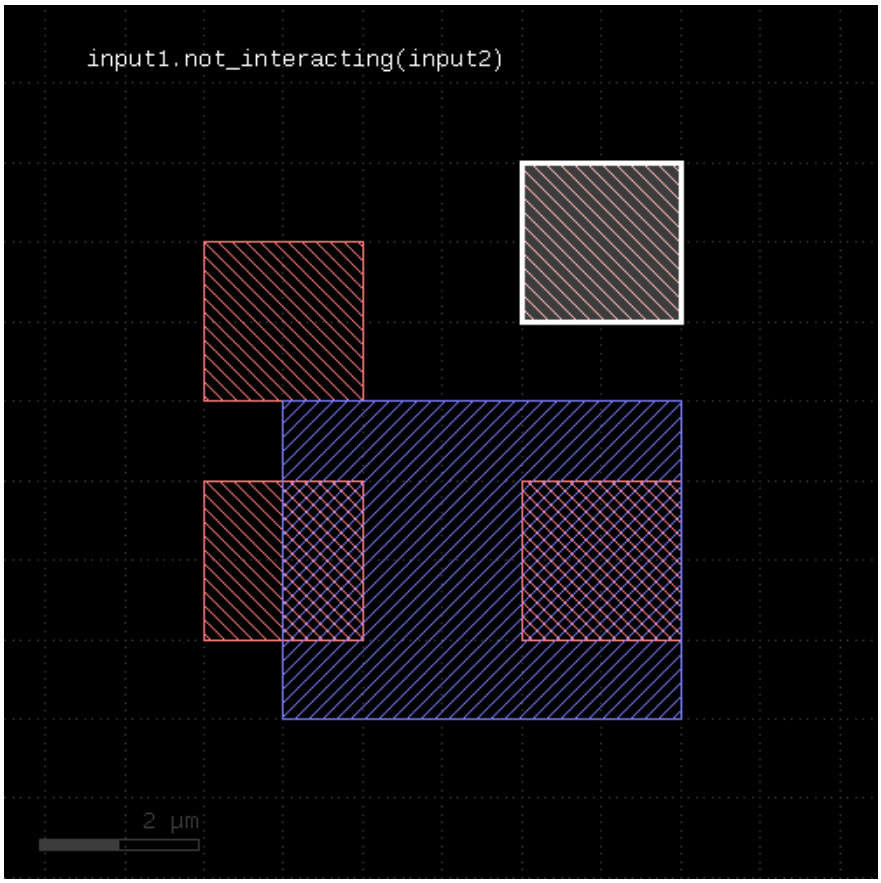
Usage:

- `layer.not_interacting(other)`
- `layer.not_interacting(other, min_count)`
- `layer.not_interacting(other, min_count, max_count)`
- `layer.not_interacting(other, min_count .. max_count)`

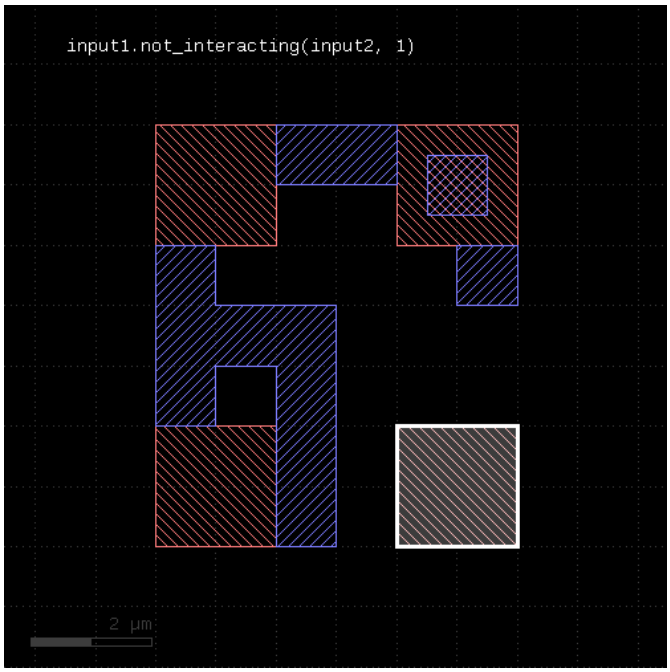
This method selects all shapes or regions from self which do not touch or overlap shapes from the other region. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. It returns a new layer containing the selected shapes. A version which modifies self is select_not_interacting.

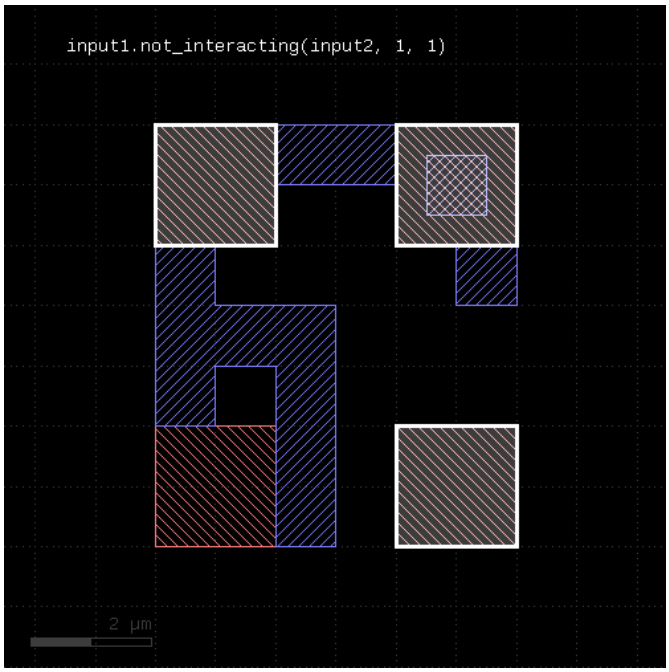
This method is available for polygon, text and edge layers. Edges can be selected with respect to other edges or polygons. Texts can be selected with respect to polygons. Polygons can be selected with respect to edges, texts and other polygons.

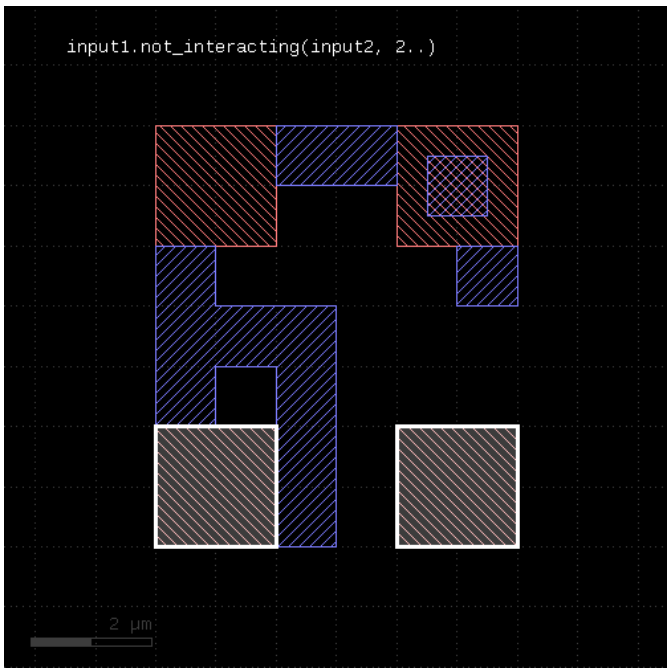
The following image shows the effect of the "not_interacting" method (input1: red, input2: blue):

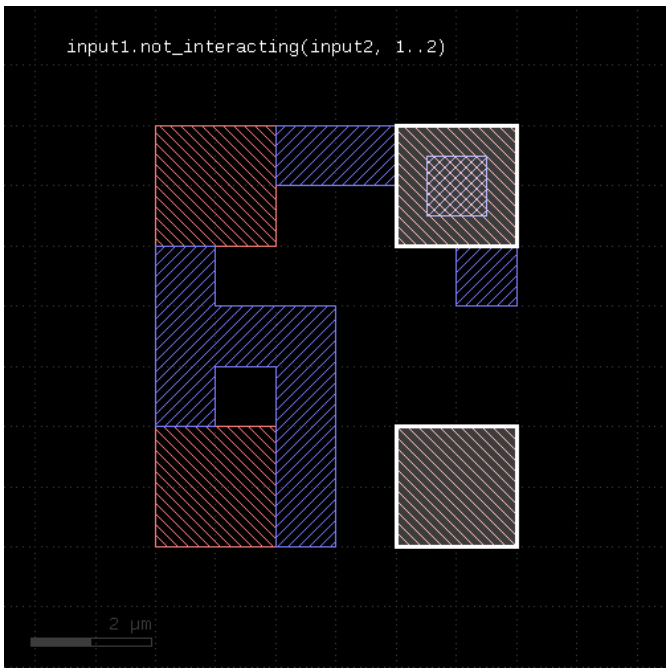


If a single count is given, shapes from self are selected only if they interact with less than the given number of (different) shapes from the other layer. If a min and max count is given, shapes from self are selected only if they interact with less than min_count or more than max_count different shapes from the other layer. Two polygons overlapping or touching at two locations are counted as single interactions.









"not_outside" - Selects edges or polygons of self which are not outside edges or polygons from the other layer

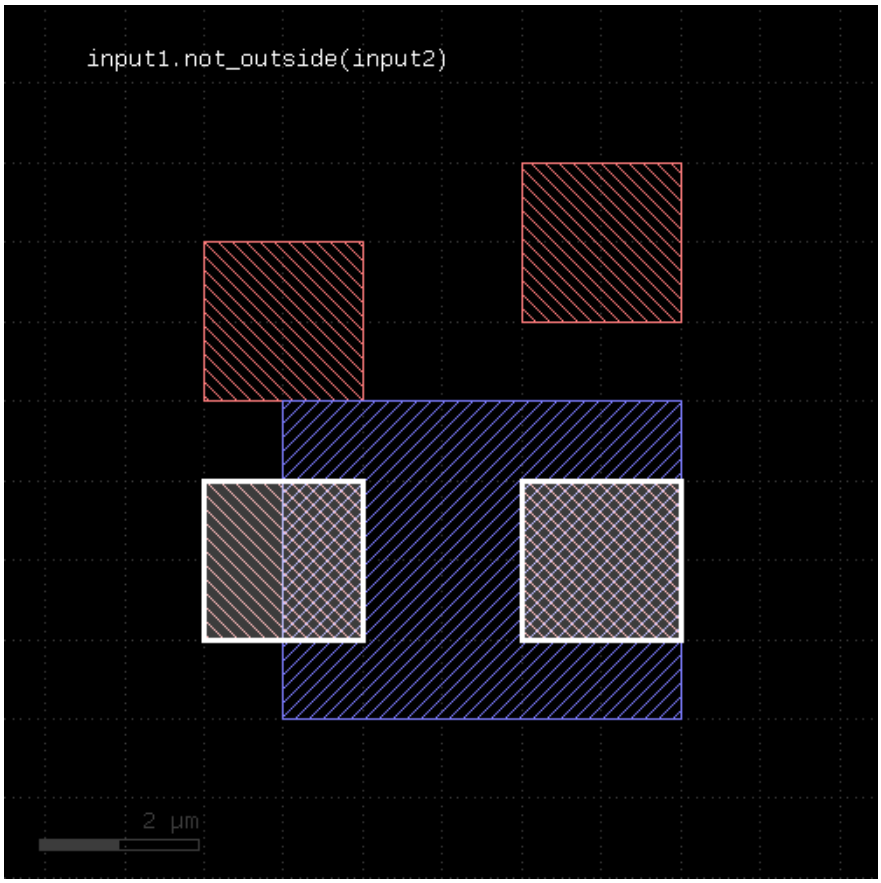
Usage:

- `layer.not_outside(other)`

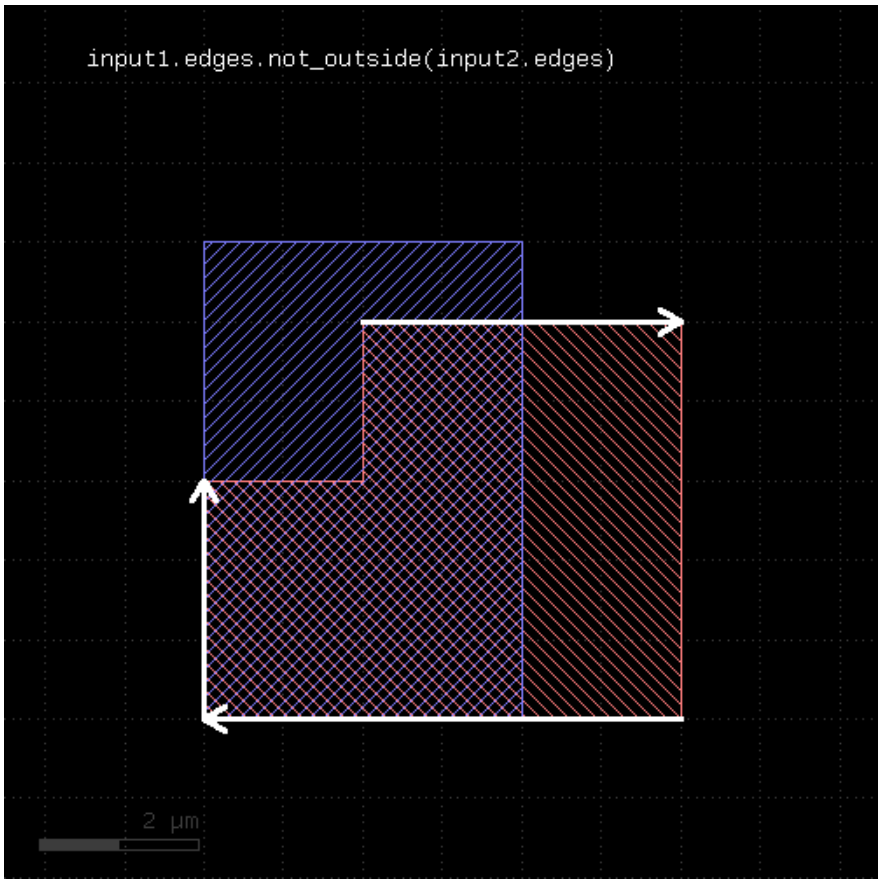
This method computes the inverse of outside - i.e. edges or polygons from the layer not being outside polygons or edges from the other layer.

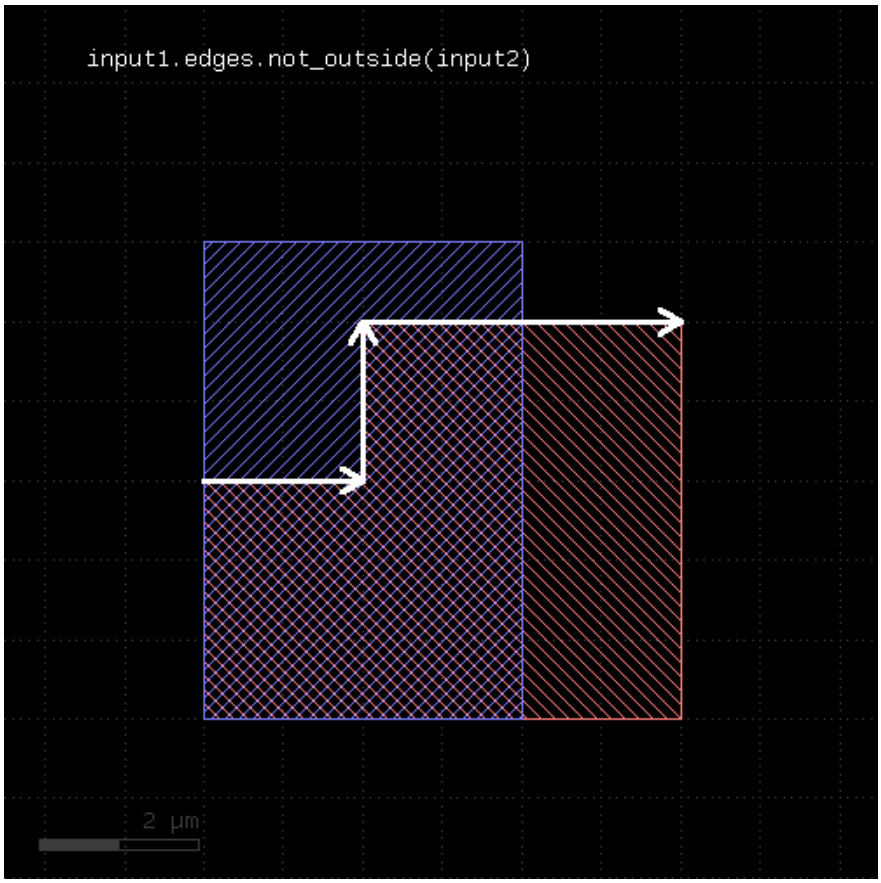
This method returns a new layer containing the selected shapes. A version which modifies self is select_not_outside. split_outside is a function computing both results of outside and not_outside in a single call. outside is a similar function selecting edges or polygons outside other edges or polygons. Note that "outside" is not the same than "not outside".

The following image shows the effect of the "not_outside" method for polygon layers (input1: red, input2: blue):



The following images show the effect of the "not_outside" method for edge layers and edge or polygon layers the second input. Note that the edges are computed from the polygons in this example (input1: red, input2: blue):


input1.edges.not_outside(input2.edges)
2 µm



"not_overlapping" - Selects shapes or regions of self which do not overlap shapes from the other region

Usage:

- `layer.not_overlapping(other)`
- `layer.not_overlapping(other, min_count)`
- `layer.not_overlapping(other, min_count, max_count)`
- `layer.not_overlapping(other, min_count .. max_count)`

This method selects all shapes or regions from self which do not overlap shapes from the other region. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. This method will return the inverse of overlapping and provides the same options.

The "not_overlapping" method is similar to the outside method. However, "outside" does not provide the option to specify counts.

This method is available for polygons only. It returns a new layer containing the selected shapes. A version which modifies self is select_not_overlapping.

"notch" - An intra-polygon spacing check

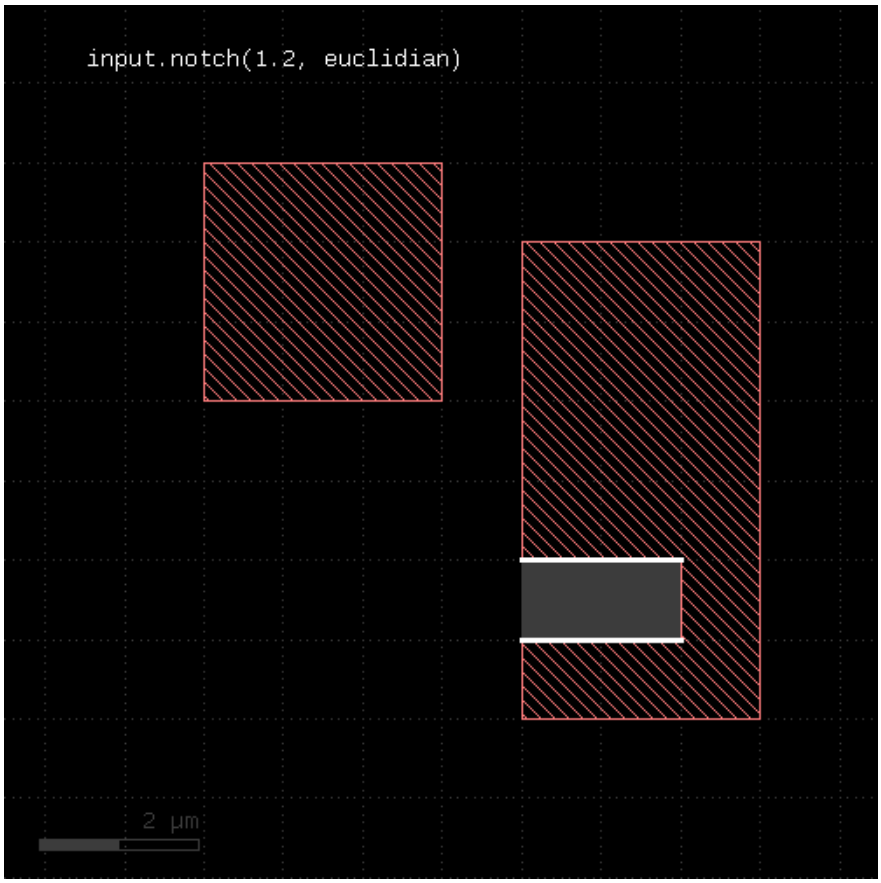
Usage:

- `layer.notch(value [, options])`

Note: "notch" is available as an operator for the "universal DRC" function Layer#drc within the DRC framework. This variant has more options and is more intuitive to use. See notch for more details.

See space for a description of this method. "notch" is the space check variant which finds space violations within a single polygon, but not against other polygons. In contrast to space, the "notch" method is available for polygon layers only, since only on such layers different polygons can be identified. Also, opposite and rectangle error filtering is not available for this method.

The following image shows the effect of the notch check:



"odd_polygons" - Checks for odd polygons (self-overlapping, non-orientable)

Usage:

- `layer.odd_polygons`

Returns the parts of the polygons which are not orientable (i.e. "8" configuration) or self-overlapping. Merged semantics does not apply for this method. Always the raw polygons are taken (see raw).

The odd_polygons check is not available in deep mode currently. See deep_reject_odd_polygons for an alternative.

"ongrid" - Checks for on-grid vertices

Usage:

- `layer.ongrid(g)`
- `layer.ongrid(gx, gy)`

Returns a single-vertex marker for each vertex whose x coordinate is not a multiple of g or gx or whose y coordinate is not a multiple of g or gy. The single-vertex markers are edge pair objects which describe a single point. When setting the grid to 0, no grid check is performed in that specific direction.

This method requires a polygon layer. Merged semantics applies (see raw and clean).

"or" - Boolean OR operation

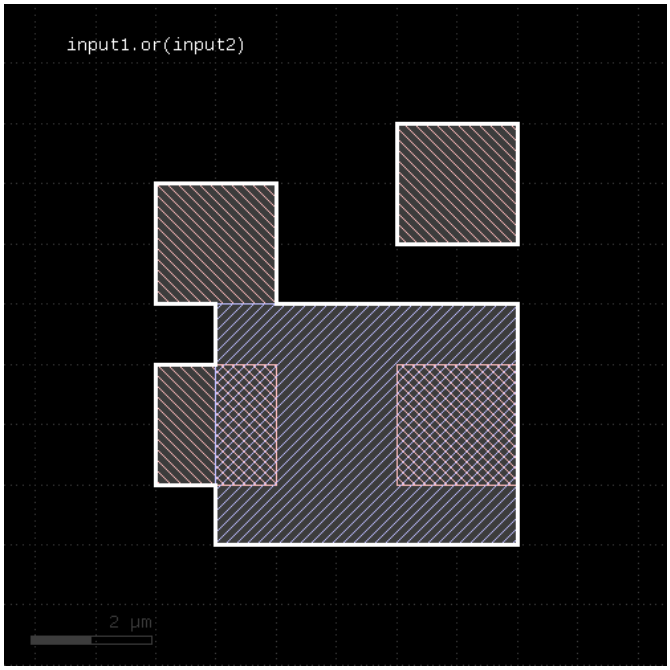
Usage:

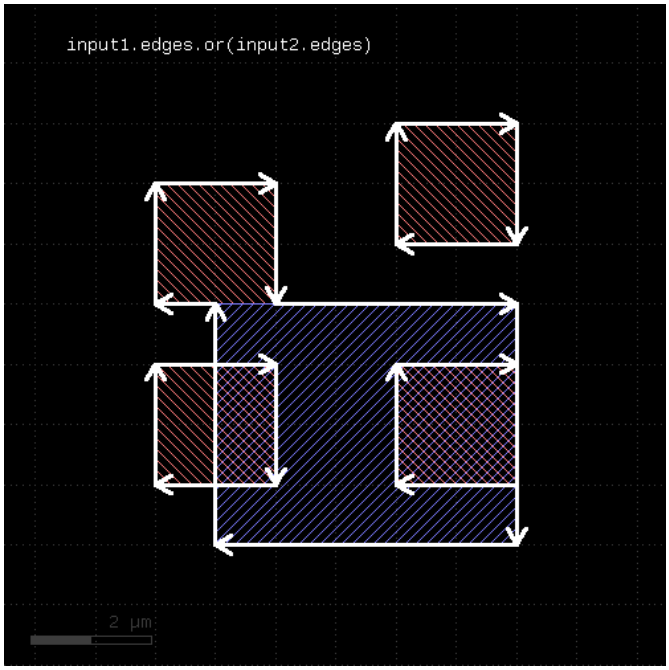
- `layer.or(other)`

The method computes a boolean OR between self and other. It is an alias for the "|" operator.

This method is available for polygon and edge layers.

The following images show the effect of the "or" method on polygons and edges (input1: red, input2: blue):





"output" - Outputs the content of the layer

Usage:

- `layer.output(specs)`

This method will copy the content of the layer to the specified output.

If a report database is selected for the output, the specification has to include a category name and optionally a category description.

If the layout is selected for the output, the specification can consist of one to three parameters: a layer number, a data type (optional, default is 0) and a layer name (optional). Alternatively, the output can be specified by a single LayerInfo object.

See report and target on how to configure output to a target layout or report database.

See also new_target and new_report on how to create additional targets for output. This allows saving certain layers to different files than the standard target or report. To do so, create a new target or report using one of these functions and pass that object to the corresponding "output" call as an additional argument.

Example:

```
check1 = ...
check2 = ...
check3 = ...

second_report = new_report("Only for check2", "check2.lyrdb")

check1.output("Check 1")
check2.output("Check 2", second_report)
check3.output("Check 3")
```

"outside" - Selects edges or polygons of self which are outside edges or polygons from the other layer

Usage:

- `layer.outside(other)`

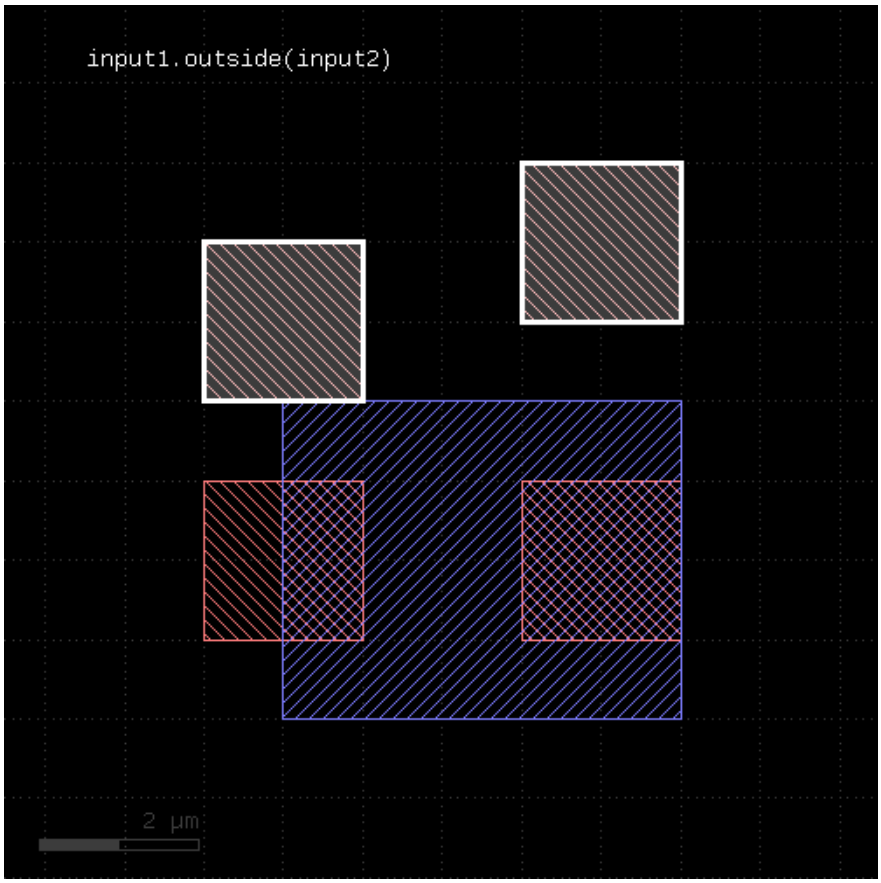
If layer is a polygon layer, the other layer needs to be a polygon layer too. In this case, this method selects all polygons which are entirely outside polygons from the other layer.

If layer is an edge layer, the other layer can be polygon or edge layer. In the first case, all edges entirely outside the polygons from the other layer are selected. If the other layer is an edge layer, all edges entirely outside of edges from the other layer are selected.

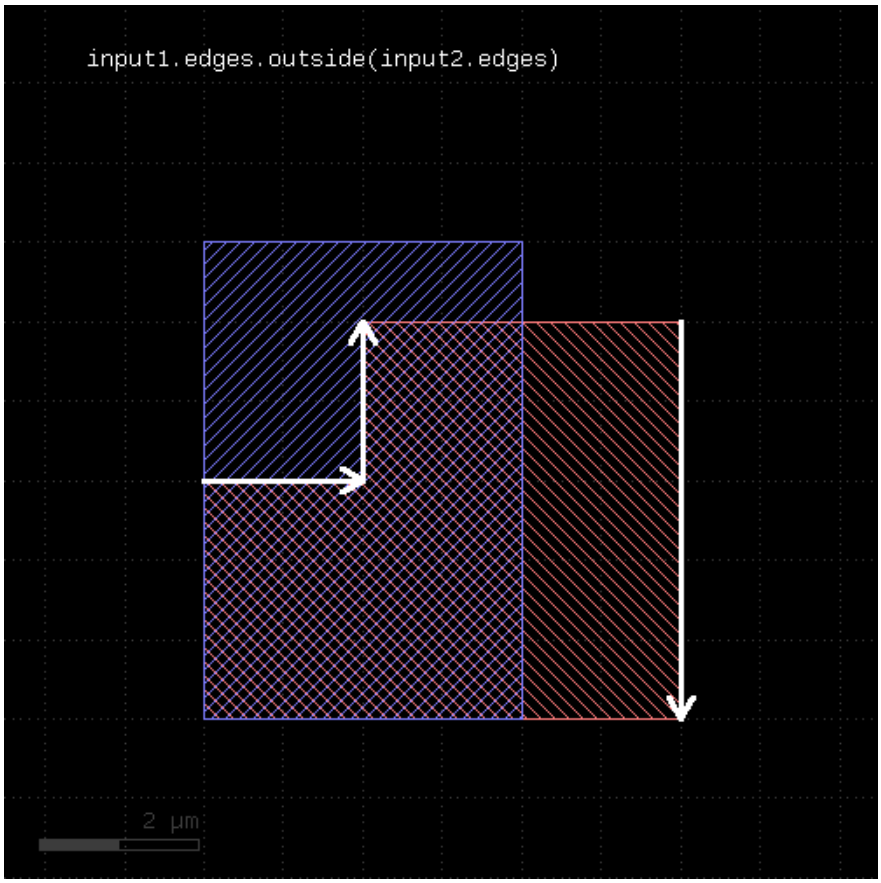
Merged semantics applies - i.e. edges or polygons are joined before the result is computed, unless the layers are in raw mode.

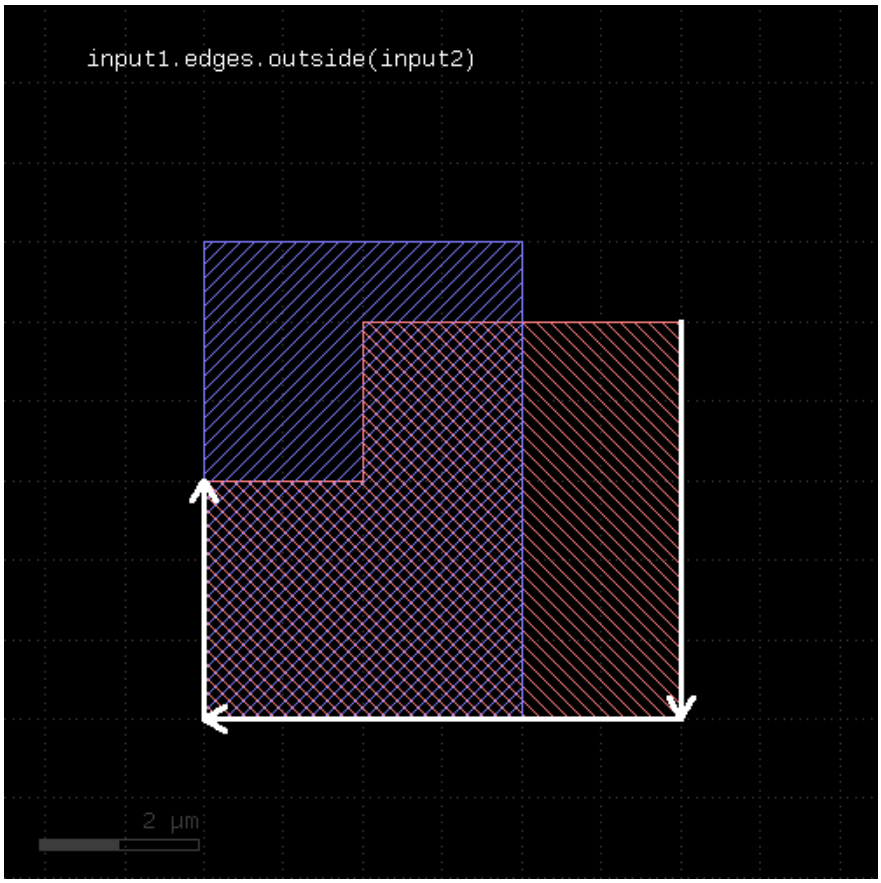
This method returns a new layer containing the selected shapes. A version which modifies self is select_outside. not_outside is a function computing the inverse of outside. split_outside is a function computing both results in a single call. outside is a similar function selecting edges or polygons outside other edges or polygons.

The following image shows the effect of the "outside" method for polygons (input1: red, input2: blue):



The following images show the effect of the "outside" method for edge layers and edge or polygon layers the second input. Note that the edges are computed from the polygons in this example (input1: red, input2: blue):


input1.edges.outside(input2.edges)
2 µm



"outside_part" - Returns the parts of the edges outside the given region

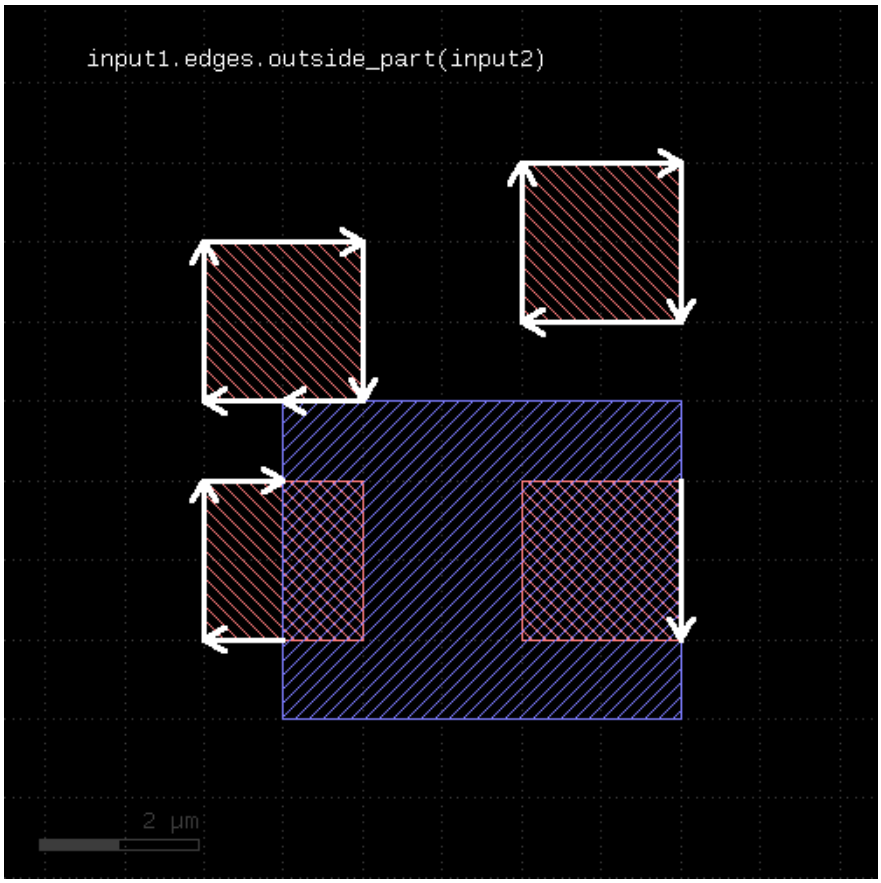
Usage:

- `layer.outside_part(region)`

This method returns the parts of the edges which are outside the given region. This is similar to the "&" operator, but this method does not remove edges that are exactly on the boundaries of the polygons of the region.

This method is available for edge layers. The argument must be a polygon layer.

inside_part is a method computing the opposite part. inside_outside_part is a method computing both inside and outside part in a single call.



"overlap" - An overlap check

Usage:

- `layer.overlap(other_layer, value [, options])`

Note: "overlap" is available as an operator for the "universal DRC" function drc within the DRC framework. This variant has more options and is more intuitive to use. See overlap for more details.

This method checks whether layer and other_layer overlap by at least the given length. Locations, where this is not the case will be reported in form of edge pair error markers. Locations, where both layers touch will be reported as errors as well. Formally such locations form an overlap with a value of 0. Locations, where both regions do not overlap or touch will not be reported. Such regions can be detected with outside or by a boolean "not".

The options are the same as for separation.

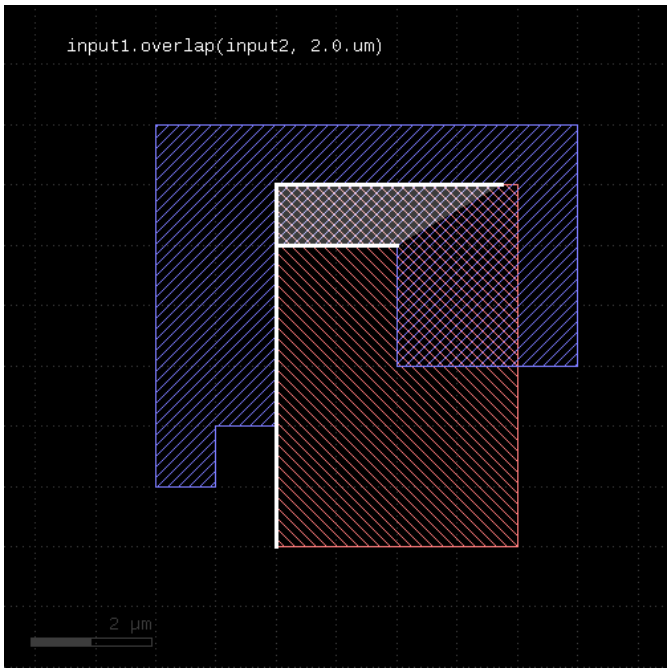
Formally, the overlap method is a two-layer width check. In contrast to the single- layer width method (width), the zero value also triggers an error and separate polygons are checked against each other, while for the single-layer width, only single polygons are considered.

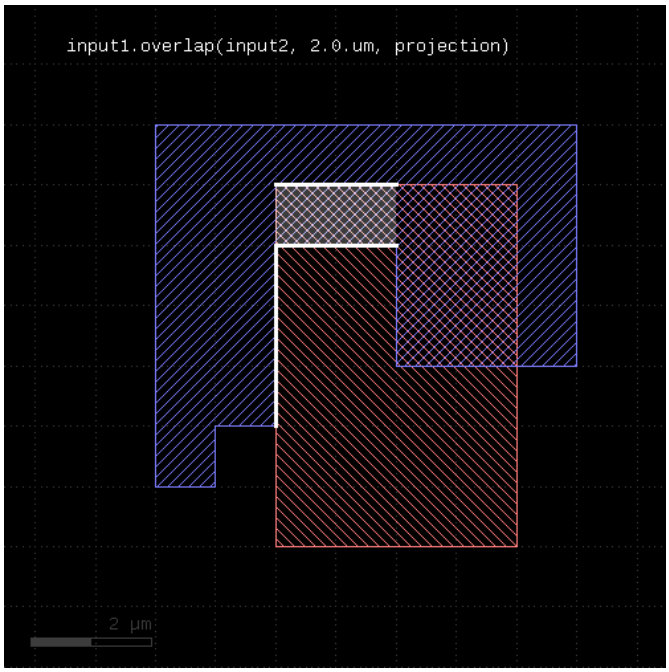
The overlap method can be applied to both edge or polygon layers. On edge layers the orientation of the edges matters: only edges which run back to back with their inside side pointing towards each other are checked for distance.

As for the other DRC methods, merged semantics applies.

Distance values can be given as floating-point values (in micron) or integer values (in database units). To explicitly specify the unit, use the unit denominators.

The following images show the effect of the overlap check (input1: red, input2: blue):





"overlapping" - Selects shapes or regions of self which overlap shapes from the other region

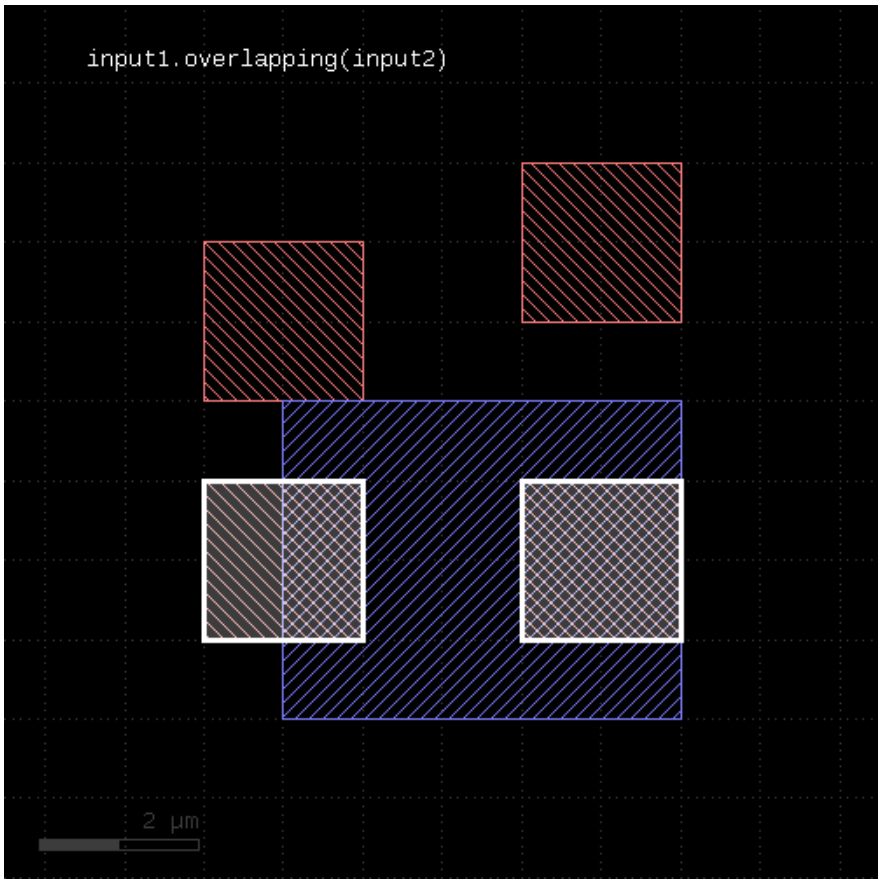
Usage:

- `layer.overlapping(other)`
- `layer.overlapping(other, min_count)`
- `layer.overlapping(other, min_count, max_count)`
- `layer.overlapping(other, min_count .. max_count)`

This method selects all shapes or regions from self which overlap shapes from the other region. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. It returns a new layer containing the selected shapes. A version which modifies self is select_overlapping.

This method is available for polygons only.

The following image shows the effect of the "overlapping" method:



A range of counts can be specified. If so, the shape from the primary layer is only selected when overlapping a given number of shapes from the other layer. For the interpretation of the count see interacting.

"perimeter" - Returns the total perimeter of the polygons in the region

Usage:

- `layer.perimeter`

This method requires a polygon layer. It returns the total perimeter of all polygons in micron. Merged semantics applies, i.e. before computing the perimeter, the polygons are merged unless raw mode is chosen (see raw).

The returned value gives the perimeter in micrometer units.

"polygons" - Returns polygons from edge pairs

Usage:

- `layer.polygons([ enlargement ])`

This method applies to edge pair collections. The edge pairs will be converted into polygons connecting the edges the edge pairs are made of. In order to properly handle special edge pairs (coincident edges, point-like edges etc.) an enlargement parameter can be specified which will make the resulting polygon somewhat larger than the original edge pair. If the enlargement parameter is 0, special edge pairs with an area of 0 will be dropped.

"polygons?" - Returns true, if the layer is a polygon layer

Usage:

- `layer.polygons?`

"pull_inside" - Selects shapes or regions of other which are inside polygons from the this region

Usage:

- `layer.pull_inside(other)`

This method selects all shapes or regions from other which are inside polygons from this region. Unless other is in raw mode (see raw), coherent regions are selected from other, otherwise individual shapes are selected.

The functionality is similar to select_inside, but chosing shapes from other rather than from self. Because in deep mode the hierarchy reference comes from self, this method provides a way to pull shapes from other to the hierarchy to self.

This method is available for polygon layers. Other needs to be a polygon layer too.

"pull_interacting" - Selects shapes or edges of other which touch or overlap shapes from the this region

Usage:

- `layer.pull_interacting(other)`

This method selects all shapes or regions from other which touch or overlap shapes from this region. Unless other is in raw mode (see raw), coherent regions are selected from other, otherwise individual shapes are selected.

The functionality is similar to select_interacting, but chosing shapes from other rather than from self. Because in deep mode the hierarchy reference comes from self, this method provides a way to pull shapes from other to the hierarchy to self.

This method will neither modify self nor other.

This method is available for polygon, edge and text layers, similar to interacting.

"pull_overlapping" - Selects shapes or regions of other which overlap shapes from the this region

Usage:

- `layer.pull_overlapping(other)`

This method selects all shapes or regions from other which overlap shapes from this region. Unless other is in raw mode (see raw), coherent regions are selected from other, otherwise individual shapes are selected.

The functionality is similar to select_overlapping, but chosing shapes from other rather than from self. Because in deep mode the hierarchy reference comes from self, this method provides a way to pull shapes from other to the hierarchy to self.

This method is available for polygon layers. Other needs to be a polygon layer too.

"raw" - Marks a layer as raw

Usage:

- `layer.raw`

A raw layer basically is the opposite of a "clean" layer (see clean). Polygons on a raw layer are considered "as is", i.e. overlapping polygons are not connected and inner edges may occur due to cut lines. Holes may not exists if the polygons are derived from a representation that does not allow holes (i.e. GDS2 files).

Note that this method will set the state of the layer. In combination with the fact, that copied layers are references to the original layer, this may lead to unexpected results:

```
l = ...
l2 = l1
... do something
l.raw
# now l2 is also a raw layer
```

To avoid that, use the dup method to create a real (deep) copy.

"rectangles" - Selects all rectangles from the input

Usage:

- `layer.rectangles`

This method is available for polygon layers. By default "merged" semantics applies, i.e. all polygons are merged before rectangles are selected (see clean and raw). non_rectangles will select all non-rectangles.

"rectilinear" - Selects all rectilinear polygons from the input

Usage:

- `layer.rectilinear`

This method is available for polygon layers. By default "merged" semantics applies, i.e. all polygons are merged before rectilinear polygons are selected (see clean and raw). non_rectilinear will select all non-rectangles.

"remove_props" - Returns a new layer with all properties removed

Usage:

- `layer.remove_props`

This method will drop all user properties from the layer. Note that a new layer without properties is returned. The original layer stays untouched.

See also select_props and map_props.

"rotate" - Rotates a layer (modifies the layer)

Usage:

- `layer.rotate(a)`

Rotates the input by the given angle (in degree). The layer that this method is called upon is modified and the modified version is returned for further processing.

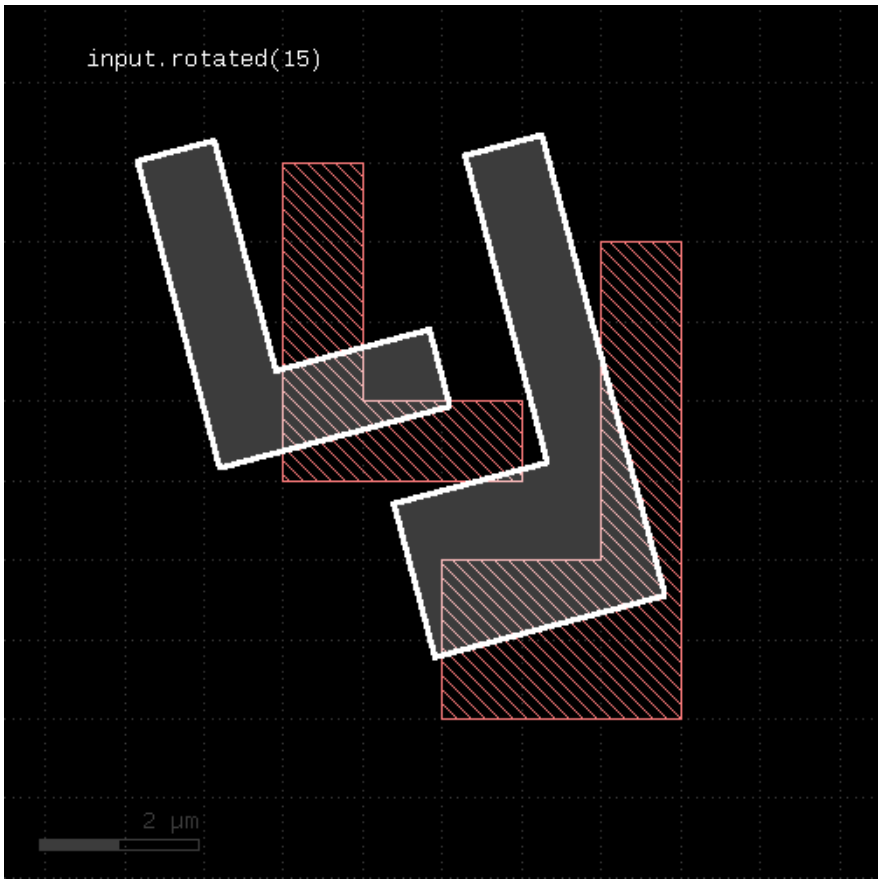
"rotated" - Rotates a layer

Usage:

- `layer.rotated(a)`

Rotates the input layer by the given angle (in degree) and returns the rotated layer. The layer that this method is called upon is not modified.

The following image shows the effect of the "rotated" method:



"rounded_corners" - Applies corner rounding to each corner of the polygon

Usage:

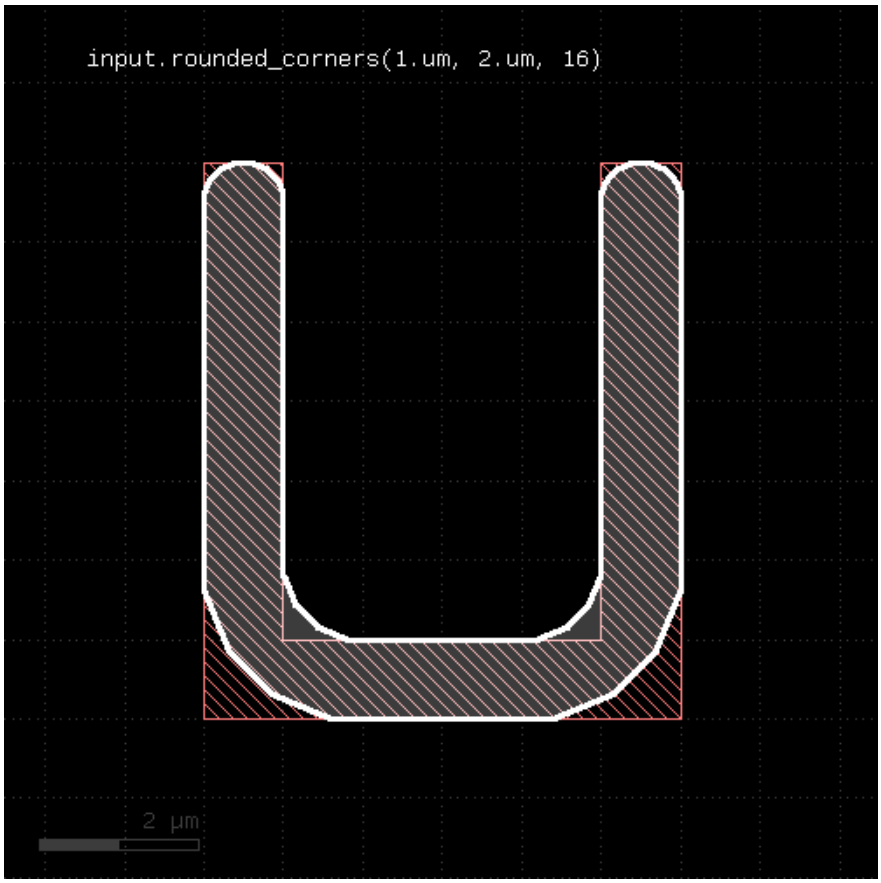
- `layer.rounded_corners(inner, outer, n)`

Inner (concave) corners are replaced by circle segments with a radius given by the "inner" parameter. Outer (convex) corners are relaced by circle segments with a radius given by the "outer" parameter.

The circles are approximated by polygons. "n" segments are used to approximate a full circle.

This method return a layer wit the modified polygons. Merged semantics applies for this method (see raw and clean). If used with tiling, the rounded_corners function may render invalid results because in tiling mode, not the whole merged region may be captured. In that case, inner edges may appear as outer ones and their corners will receive rounding.

The following image shows the effect of the "rounded_corners" method. The upper ends of the vertical bars are rounded with a smaller radius automatically because their width does not allow a larger radius.



"scale" - Scales a layer (modifies the layer)

Usage:

- `layer.scale(f)`

Scales the input. After scaling, features have a f times bigger dimension. The layer that this method is called upon is modified and the modified version is returned for further processing.

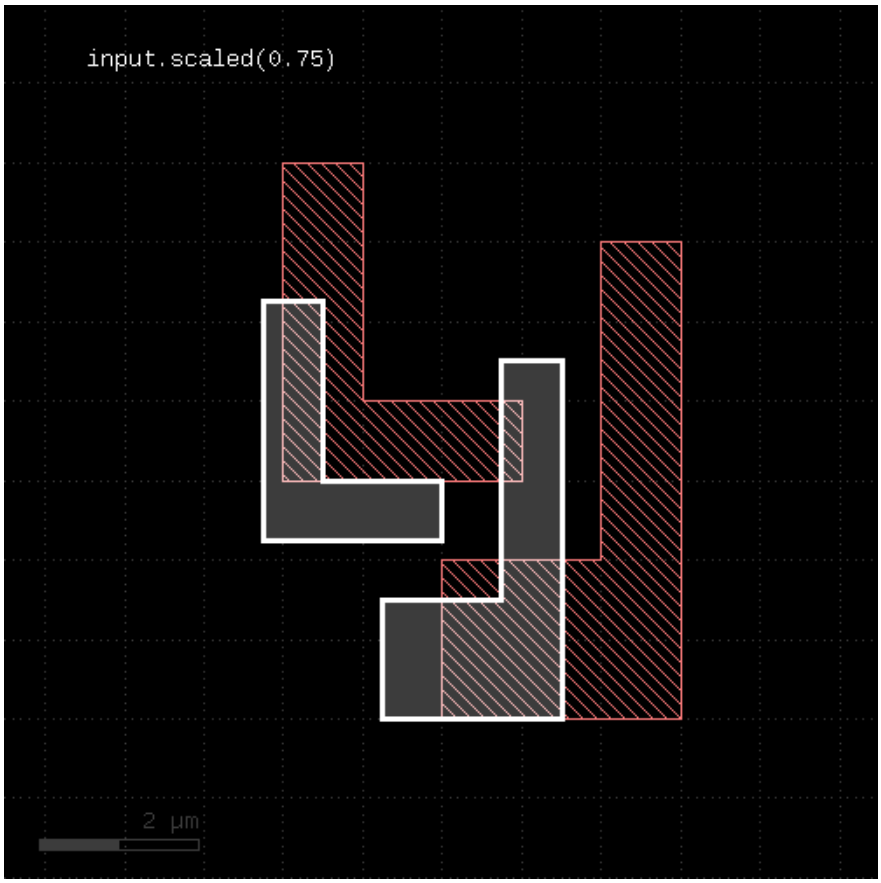
"scaled" - Scales a layer

Usage:

- `layer.scaled(f)`

Scales the input layer and returns a new layer whose features have a f times bigger dimension. The layer that this method is called upon is not modified.

The following images shows the effect of the "scaled" method:



"second_edges" - Returns the second edges of an edge pair collection

Usage:

- `layer.second_edges`

Applies to edge pair collections only. Returns the second edges of the edge pairs in the collection.

Some checks deliver symmetric edge pairs (e.g. space, width, etc.) for which the edges are commutable. "second_edges" will not deliver edges for such edge pairs. Instead, "first_edges" will deliver both.

"select" - Selects edges, edge pairs or polygons based on evaluation of a block

Usage:

- `layer.select { |object| ... }`

This method evaluates the block and returns a new container with those objects for which the block evaluates to true. It is available for edge, polygon and edge pair layers. The corresponding objects are DPolygon, DEdge or DEdgePair.

Because this method executes inside the interpreter, it's inherently slow. Tiling does not apply to this method.

Here is a (slow) equivalent of the area selection method:

```
new_layer = layer.select { |polygon| polygon.area >= 10.0 }
```

"select_covering" - Selects shapes or regions of self which completely cover (enclose) one or more shapes from the other region

Usage:

- `layer.select_covering(other)`
- `layer.select_covering(other, min_count)`
- `layer.select_covering(other, min_count, max_count)`
- `layer.select_covering(other, min_count .. max_count)`

This method selects all shapes or regions from self which cover shapes from the other region. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. It modifies self to contain the selected shapes. A version which does not modify self is covering.

This method is available for polygons only.

"select_inside" - Selects edges or polygons of self which are inside edges or polygons from the other layer

Usage:

- `layer.select_inside(other)`

This method is the in-place version of inside - i.e. it modifies the layer instead of returning a new layer and leaving the original layer untouched.

"select_interacting" - Selects shapes or regions of self which touch or overlap shapes from the other region

Usage:

- `layer.select_interacting(other)`
- `layer.select_interacting(other, min_count)`
- `layer.select_interacting(other, min_count, max_count)`
- `layer.select_interacting(other, min_count .. max_count)`

This method selects all shapes or regions from self which touch or overlap shapes from the other layer. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. It modifies self to contain the selected shapes. A version which does not modify self is interacting.

This method is available for polygon, text and edge layers. Edges can be selected with respect to other edges or polygons. Texts can be selected with respect to polygons. Polygons can be selected with respect to edges, texts and other polygons.

If a single count is given, shapes from self are selected only if they do interact at least with the given number of (different) shapes from the other layer. If a min and max count is given, shapes from self are selected only if they interact with min_count or more, but a maximum of max_count different shapes from the other layer. Two polygons overlapping or touching at two locations are counted as single interactions.

"select_not_covering" - Selects shapes or regions of self which do not cover (enclose) one or more shapes from the other region

Usage:

- `layer.select_not_covering(other)`
- `layer.select_not_covering(other, min_count)`

- `layer.select_not_covering(other, min_count, max_count)`
- `layer.select_not_covering(other, min_count .. max_count)`

This method selects all shapes or regions from self which do not cover shapes from the other region. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. It modifies self to contain the selected shapes. A version which does not modify self is not_covering.

This method is available for polygons only.

"select_not_inside" - Selects edges or polygons of self which are not inside edges or polygons from the other layer

Usage:

- `layer.select_not_inside(other)`

This method is the in-place version of inside - i.e. it modifies the layer instead of returning a new layer and leaving the original layer untouched.

"select_not_interacting" - Selects shapes or regions of self which do not touch or overlap shapes from the other region

Usage:

- `layer.select_not_interacting(other)`
- `layer.select_not_interacting(other, min_count)`
- `layer.select_not_interacting(other, min_count, max_count)`
- `layer.select_not_interacting(other, min_count .. max_count)`

This method selects all shapes or regions from self which do not touch or overlap shapes from the other layer. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. It modifies self to contain the selected shapes. A version which does not modify self is not_interacting.

This method is available for polygon, text and edge layers. Edges can be selected with respect to other edges or polygons. Texts can be selected with respect to polygons. Polygons can be selected with respect to edges, texts and other polygons.

If a single count is given, shapes from self are selected only if they interact with less than the given number of (different) shapes from the other layer. If a min and max count is given, shapes from self are selected only if they interact with less than min_count or more than max_count different shapes from the other layer. Two polygons overlapping or touching at two locations are counted as single interactions.

"select_not_outside" - Selects edges or polygons of self which are not outside edges or polygons from the other layer

Usage:

- `layer.select_not_outside(other)`

This method is the in-place version of outside - i.e. it modifies the layer instead of returning a new layer and leaving the original layer untouched.

"select_not_overlapping" - Selects shapes or regions of self which do not overlap shapes from the other region

Usage:

- `layer.select_not_overlapping(other)`
- `layer.select_not_overlapping(other, min_count)`
- `layer.select_not_overlapping(other, min_count, max_count)`
- `layer.select_not_overlapping(other, min_count .. max_count)`

This method selects all shapes or regions from self which do not overlap shapes from the other region. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. It modifies self to contain the selected shapes. A version which does not modify self is not_overlapping.

This method is available for polygons only.

"select_outside" - Selects edges or polygons of self which are outside edges or polygons from the other layer

Usage:

- `layer.select_outside(other)`

This method is the in-place version of outside - i.e. it modifies the layer instead of returning a new layer and leaving the original layer untouched.

"select_overlapping" - Selects shapes or regions of self which overlap shapes from the other region

Usage:

- `layer.select_overlapping(other)`
- `layer.select_overlapping(other, min_count)`
- `layer.select_overlapping(other, min_count, max_count)`
- `layer.select_overlapping(other, min_count .. max_count)`

This method selects all shapes or regions from self which overlap shapes from the other region. Unless self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected. It modifies self to contain the selected shapes. A version which does not modify self is overlapping.

This method is available for polygons only.

"select_props" - Enables or selects properties from a property-annotated layer

Usage:

- `layer.select_props(keys)`

This method will select specific property keys from layers. It returns a new layer with the new properties. The original layer is not modified.

You can specify the user property keys (names) to use. As user properties in general are a set of key/value pairs and may carry multiple values under different keys, this feature can be handy to filter out a specific aspect. To get only the values from key 1 (integer), use:

```
layer1 = input(1, 0, enable_properties)
layer1_filtered = layer1.select_props(1)
```

To get the combined key 1 and 2 properties, use:

```
layer1 = input(1, 0, enable_properties)
```

```
layer1_filtered = layer1.select_props(1, 2)
```

Without any arguments, this method will remove all properties. Note that you can directly filter or map properties on input which is more efficient than first loading all and then selecting some properties. See DRCSource#input for details.

map_props is a way to change property keys and remove_props will entirely remove all user properties.

"sep" - An alias for "separation"

Usage:

- `layer.sep(value [, options])`

See separation for a description of that method

"separation" - A two-layer spacing check

Usage:

- `layer.separation(other_layer, value [, options])`
- `layer.sep(other_layer, value [, options])`

Note: "separation" and "sep" are available as operators for the "universal DRC" function drc within the DRC framework. These variants have more options and are more intuitive to use. See separation for more details.

This method performs a two-layer spacing check. Like space, this method can be applied to edge or polygon layers. Locations where edges of the layer are closer than the specified distance to the other layer are reported as edge pair error markers.

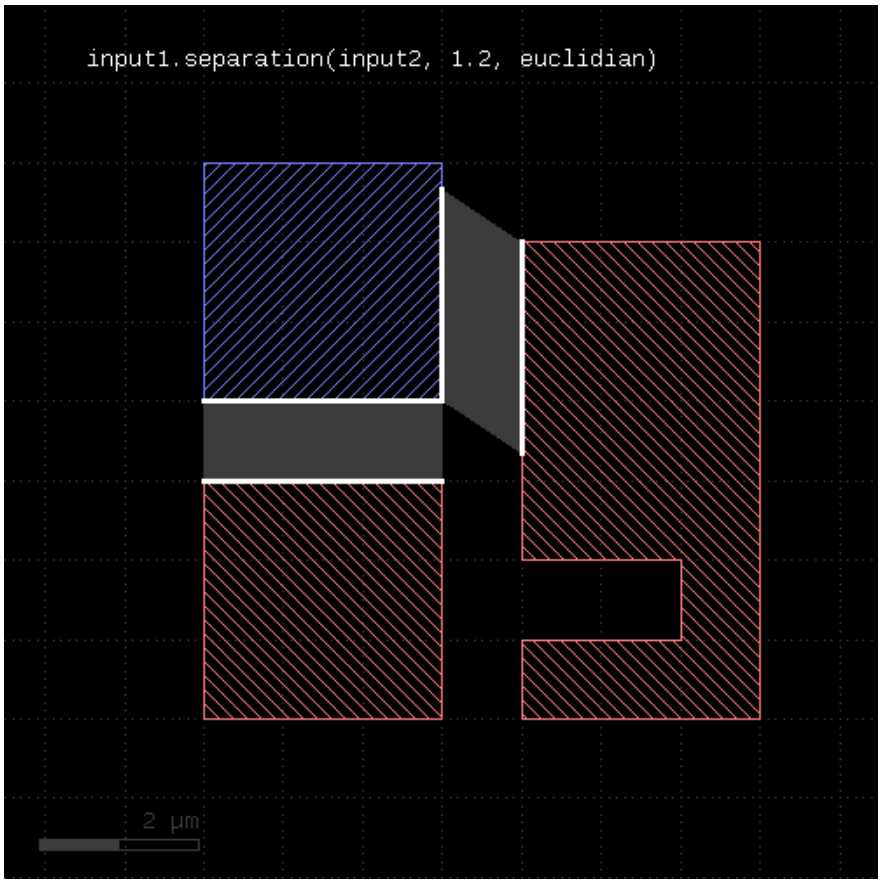
"sep" is the short form of this method.

In contrast to the space and related methods, locations where both layers touch are also reported. More specifically, the case of zero spacing will also trigger an error while for space it will not.

As for the other DRC methods, merged semantics applies.

Distance values can be given as floating-point values (in micron) or integer values (in database units). To explicitly specify the unit, use the unit denominators.

The following image shows the effect of the separation check (input1: red, input2: blue):

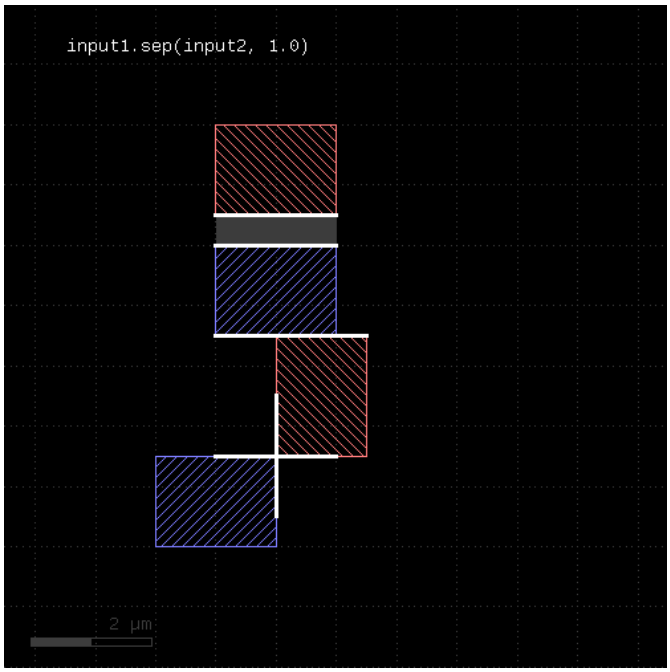


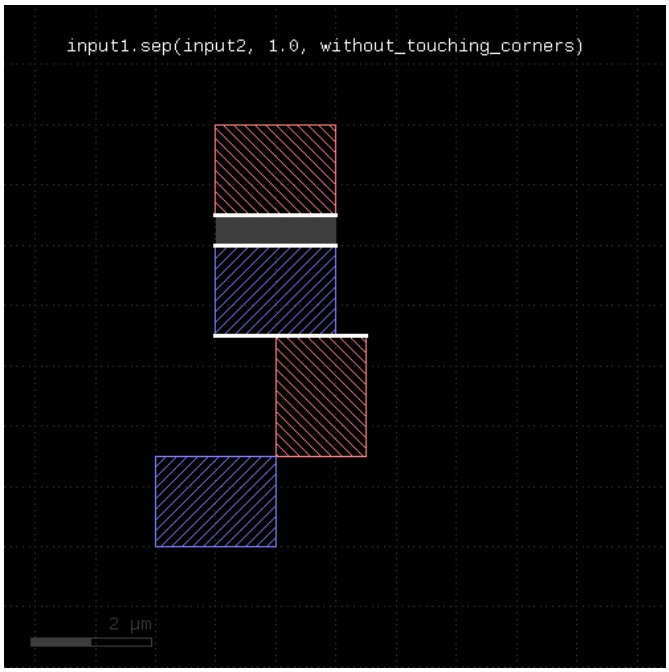
Touching shapes

Like width and space, the separation check also supports the "without_touching_corners" option.

This option will turn off errors that arise due to edges touching in one corner (the "kissing corners" configuration). By default, such edges will yield an error, as they form a zero-distance situation. With this option in place, no errors will be reported.

The following images illustrate the effect of the "without_touching_corners" option. The white line at the top of the bottom red shape is actually an edge pair indicating the zero-distance violation of the separation check:

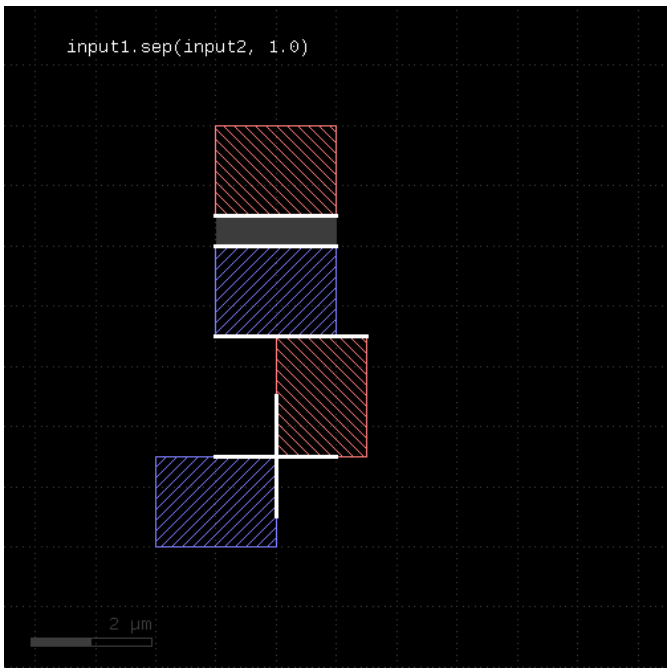


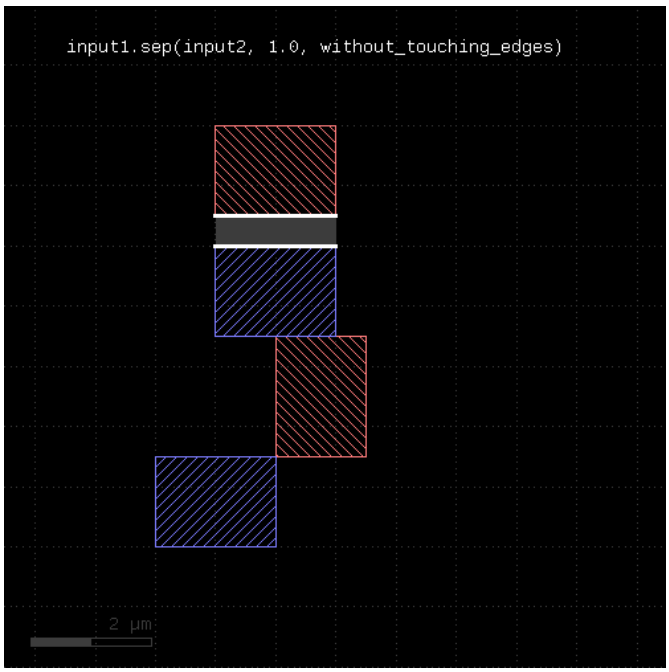


Another option is "without_touching_edges" which turns off errors that arise at coincident edges. Formally such edges represent a zero-distance situation, hence are flagged by default. Turning off the check in this case can be helpful when separating a layer into two parts (e.g. thin/wide metal separation) and an error between touching regions is not desired.

The "without_touching_edges" option is a stronger version of "without_touching_corners" and makes sense only for two-layer checks.

The following images illustrate the effect of the "without_touching_edges" option:





Opposite and rectangle error filtering

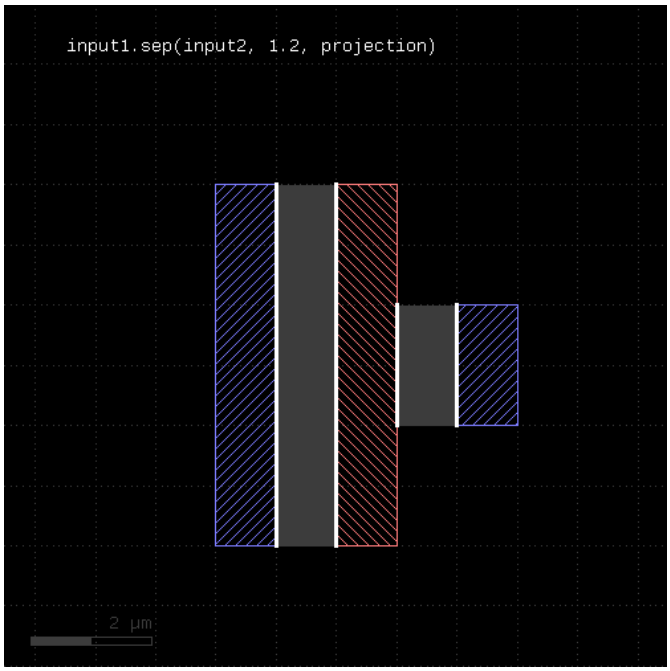
The options for the separation check are those available for the width or space method plus opposite and rectangle error filtering.

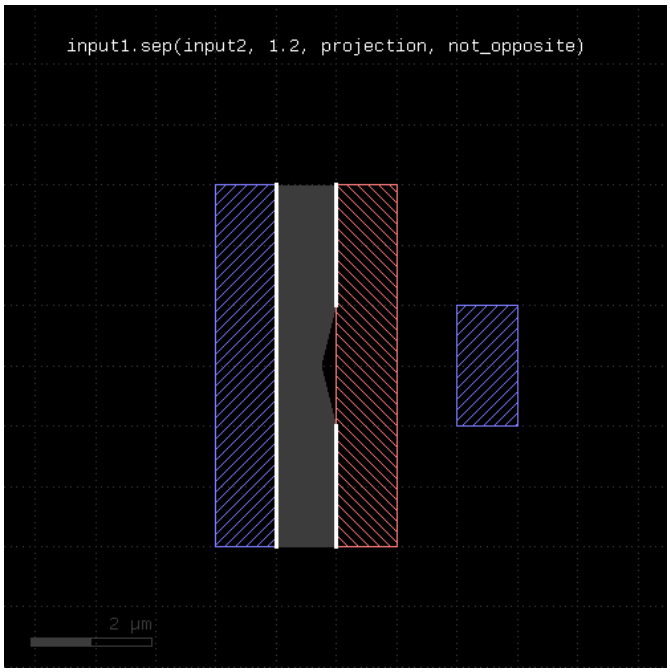
Opposite error filtering will waive errors that are on opposite sides of the original figure. The inverse is selection of errors only when there is an error present on the opposite side of the original figure. Opposite error waiving or selection is achieved through these options inside the DRC function call:

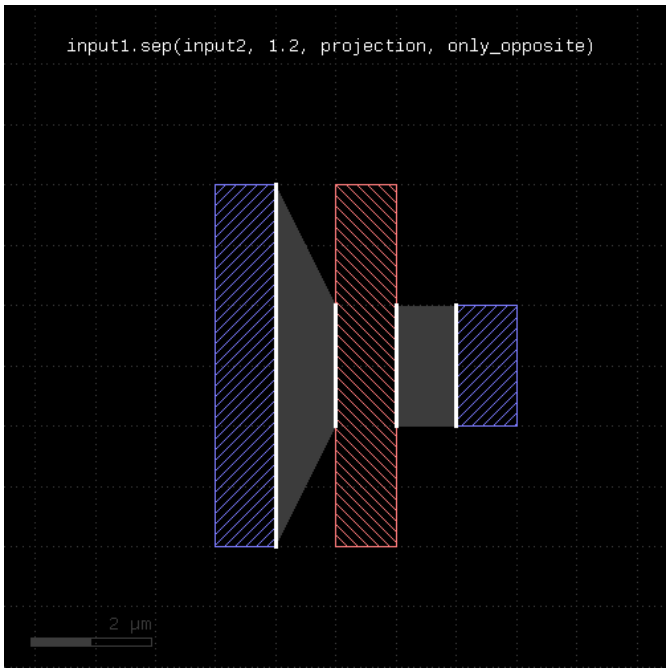
- **not_opposite** will waive opposite errors
- **only_opposite** will select errors only if there is an opposite one

These modes imply partial waiving or selection if "opposite" only applies to a section of an error.

The following images shows the effect of these options:







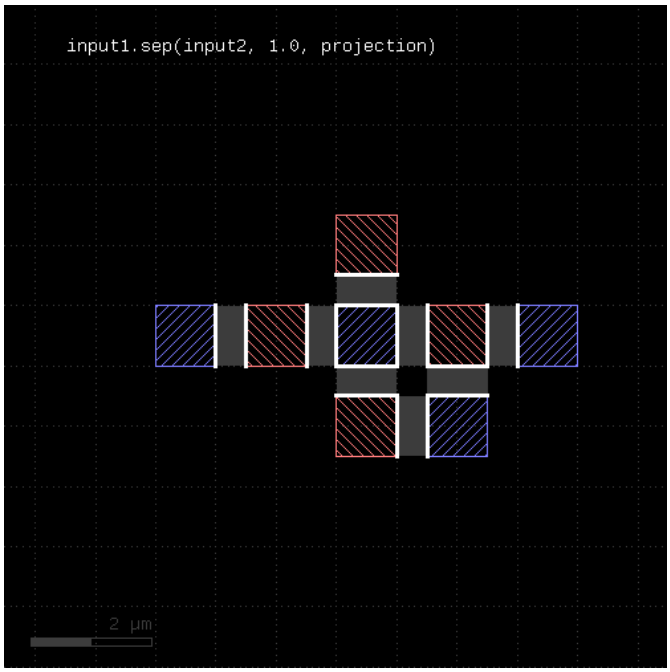
Rectangle error filtering allows waiving errors based on how they cover the sides of an original rectangular figure. This selection only applies to errors covering the full edge of the rectangle. Errors covering parts of the rectangle edges are not considered in this scheme.

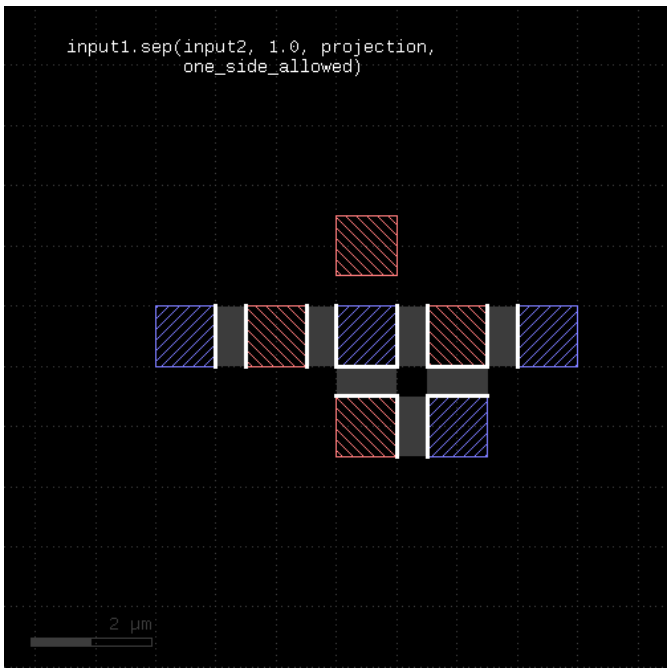
The rectangle filter option is enabled by these modes:

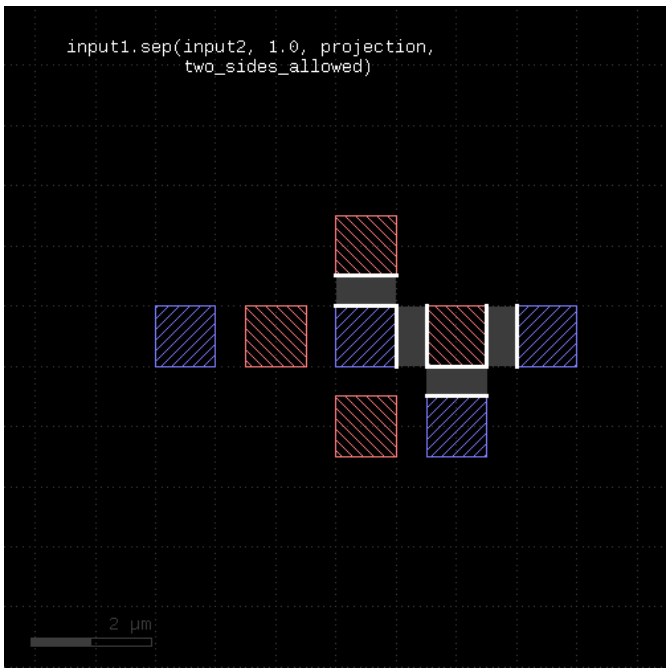
- **one_side_allowed** will waive errors when they appear on one side of the rectangle only
- **two_sides_allowed** will waive errors when they appear on two sides of the rectangle
- **two_connected_sides_allowed** will waive errors when they appear on two connected sides of the rectangle ("L" configuration)
- **two_opposite_sides_allowed** will waive errors when they appear on two opposite sides of the rectangle
- **three_sides_allowed** will waive errors when they appear on three sides of the rectangle
- **four_sides_allowed** will waive errors when they appear on four sides of the rectangle

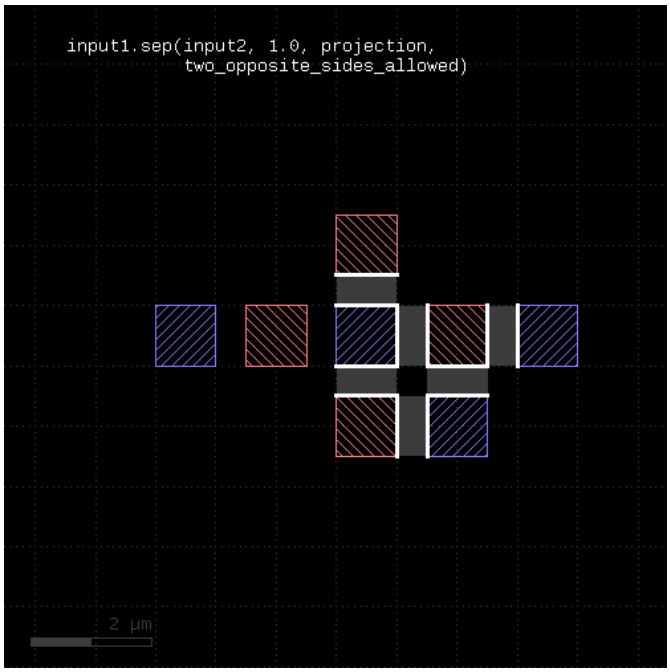
Multiple of these options can be given, which will make errors waived if one of these conditions is met.

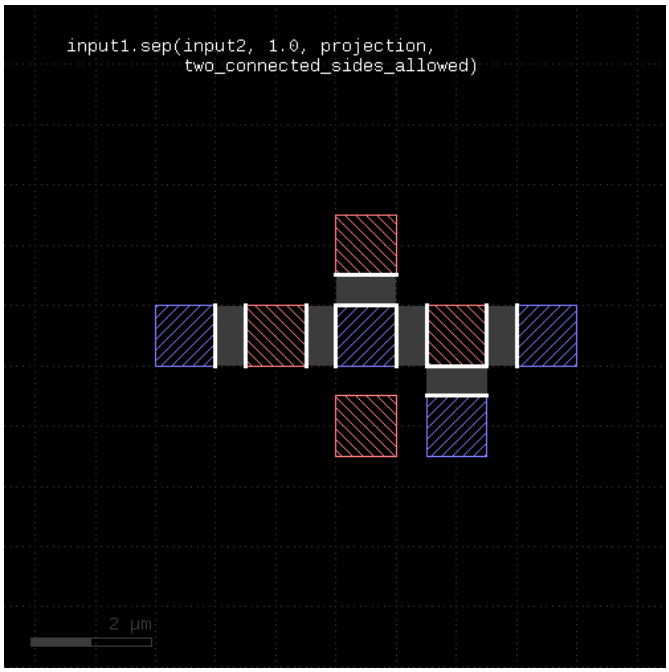
The following images shows the effect of some rectangle filter modes:

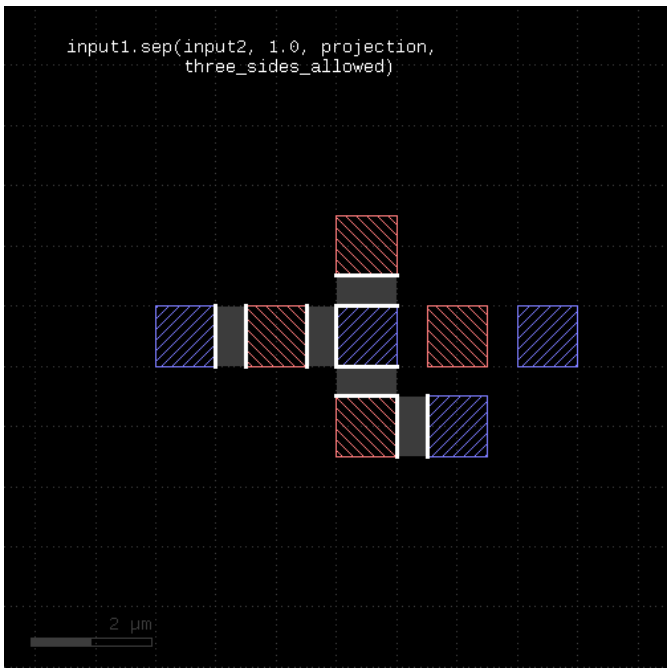

input1.sep(input2, 1.0, projection)
2 µm

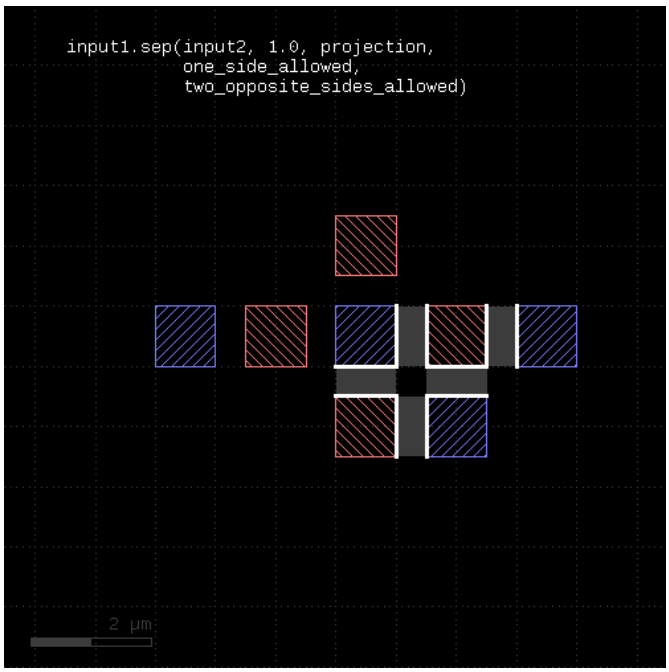

input1.sep(input2, 1.0, projection,
one_side_allowed)
2 µm


input1.sep(input2, 1.0, projection,
two_sides_allowed)
2 µm









"size" - Polygon sizing (per-edge biasing, modifies the layer)

Usage:

- `layer.size(d [, mode])`

- `layer.size(dx, dy [, mode]))`

See sized. The size method basically does the same but modifies the layer it is called on. The input layer is returned and available for further processing.

"sized" - Polygon sizing (per-edge biasing)

Usage:

- `layer.sized(d [, mode])`
- `layer.sized(dx, dy [, mode]))`

This method requires a polygon layer. It will apply a bias per edge of the polygons and return the biased layer. The layer that this method is called on is not modified.

In the single-value form, that bias is applied both in horizontal or vertical direction. In the two-value form, the horizontal and vertical bias can be specified separately.

The mode defines how to handle corners. The following modes are available:

- **diamond_limit** : This mode will connect the shifted edges without corner interpolation
- **octagon_limit** : This mode will create octagon-shaped corners
- **square_limit** : This mode will leave 90 degree corners untouched but cut off corners with a sharper angle. This is the default mode.
- **acute_limit** : This mode will leave 45 degree corners untouched but cut off corners with a sharper angle
- **no_limit** : This mode will not cut off (only at extremely sharp angles

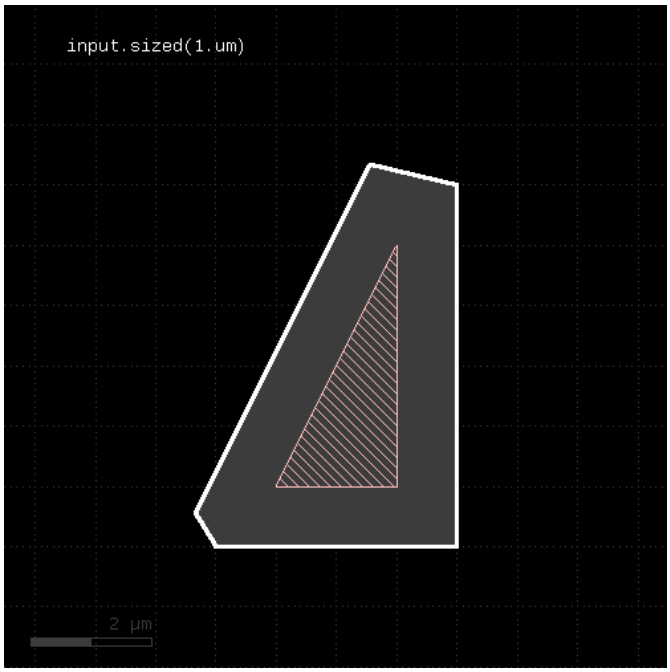
Merged semantics applies, i.e. polygons will be merged before the sizing is applied unless the layer was put into raw mode (see raw).
On output, the polygons are not merged immediately, so it is possible to detect overlapping regions after a positive sizing using raw and merged with an overlap count, for example:

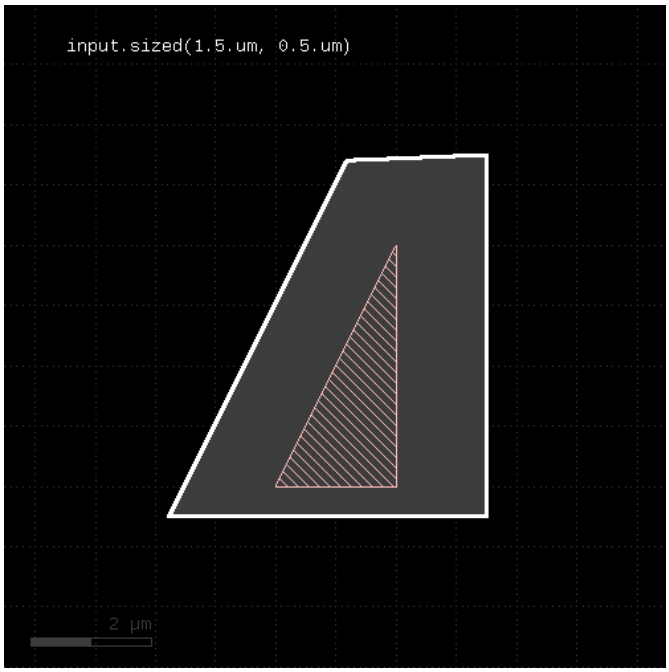
```
layer.sized(300.nm).raw.merged(2)
```

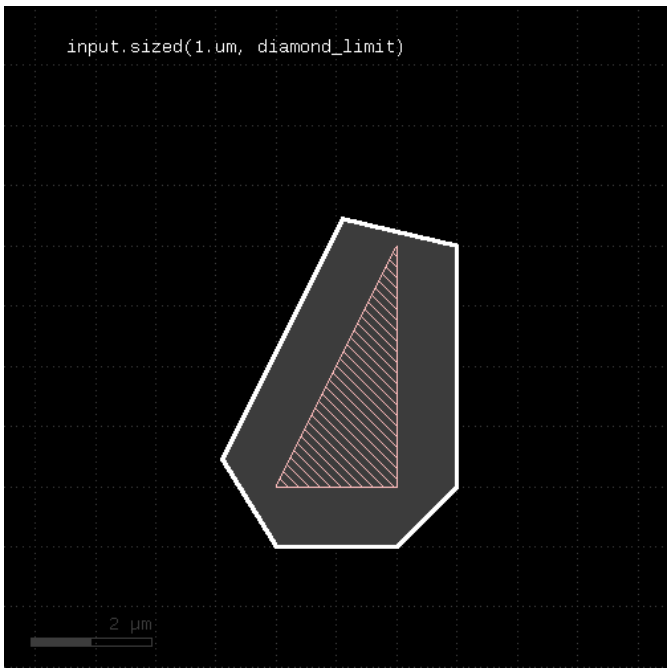
Bias values can be given as floating-point values (in micron) or integer values (in database units). To explicitly specify the unit, use the unit denominators.

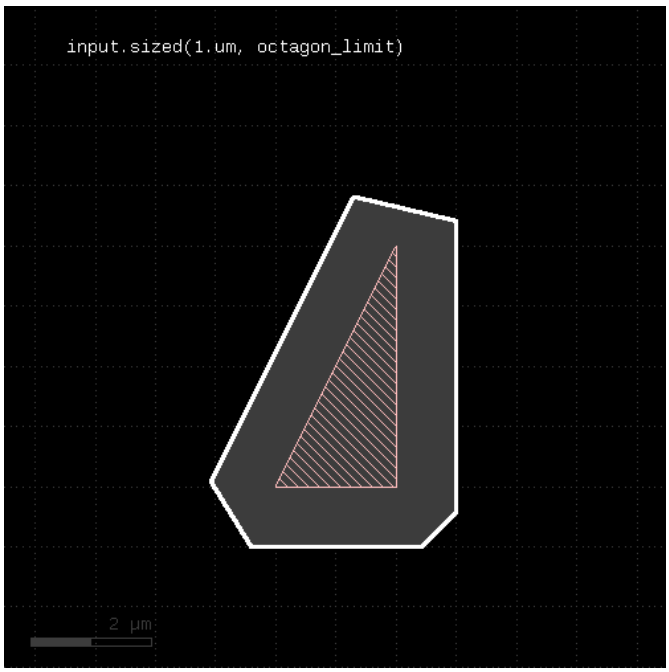
size is working like sized but modifies the layer it is called on.

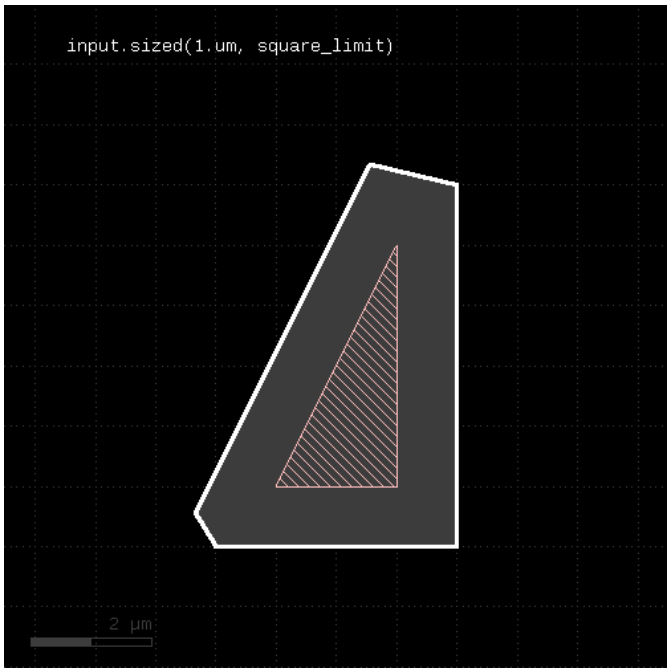
The following images show the effect of various forms of the "sized" method:

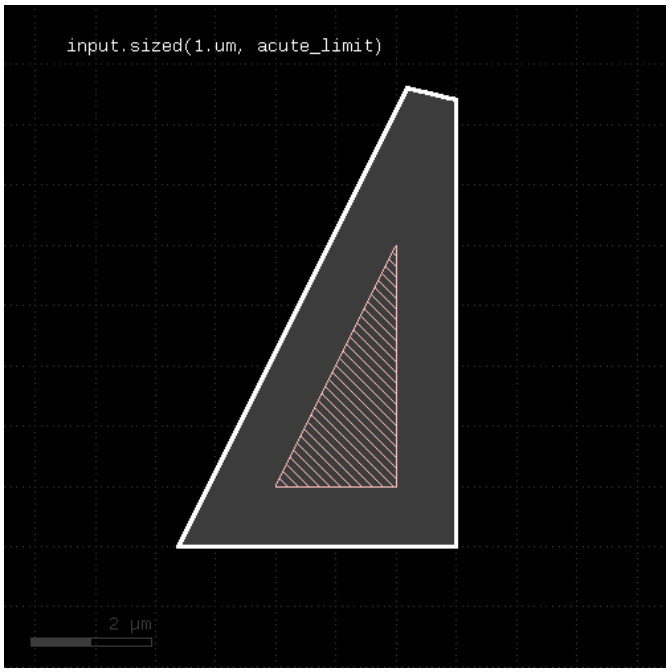

input.sized(1.um)
2 µm


input.sized(1.5.um, 0.5.um)
2 µm


input.sized(1.um, diamond_limit)
2 µm


input.sized(1.um, octagon_limit)
2 µm





"smoothed" - Smoothes the polygons of the region

Usage:

- `layer.smoothed(d)`
- `layer.smoothed(d, hv_keep)`

"Smoothing" returns a simplified version of the polygons. Simplification is achieved by removing vertices unless the resulting polygon deviates by more than the given distance d from the original polygon.

"hv_keep" is a boolean parameter which makes the smoothing function maintain horizontal or vertical edges. The default is false, meaning horizontal or vertical edges may be changed into tilted ones.

This method return a layer wit the modified polygons. Merged semantics applies for this method (see raw and clean).

"snap" - Brings each vertex on the given grid (g or gx/gy for x or y direction)

Usage:

- `layer.snap(g)`
- `layer.snap(gx, gy)`

Shifts each off-grid vertex to the nearest on-grid location. If one grid is given, this grid is applied to x and y coordinates. If two grids are given, gx is applied to the x coordinates and gy is applied to the y coordinates. If 0 is given as a grid, no snapping is performed in that direction.

This method modifies the layer. A version that returns a snapped version of the layer without modifying the layer is snapped.

This method requires a polygon layer. Merged semantics applies (see raw and clean).

"snapped" - Returns a snapped version of the layer

Usage:

- `layer.snapped(g)`
- `layer.snapped(gx, gy)`

See snap for a description of the functionality. In contrast to snap, this method does not modify the layer but returns a snapped copy.

"space" - A space check

Usage:

- `layer.space(value [, options])`

Note: "space" is available as an operator for the "universal DRC" function Layer#drc within the DRC framework. This variant has more options and is more intuitive to use. See space for more details.

This method performs a space check and returns a collection of edge pairs. A space check can be performed on polygon and edge layers. On edge layers, all edges are checked against all other edges. If two edges form a "face to face" relation (i.e. their outer sides face each other) and their distance is less than the specified value, an error shape is generated for that edge pair. On polygon layers, the polygons on each layer are checked for space against other polygons for locations where their space is less than the specified value. In that case, an edge pair error shape is generated. The space check will also check the polygons for space violations against themselves, i.e. notches violating the space condition are reported.

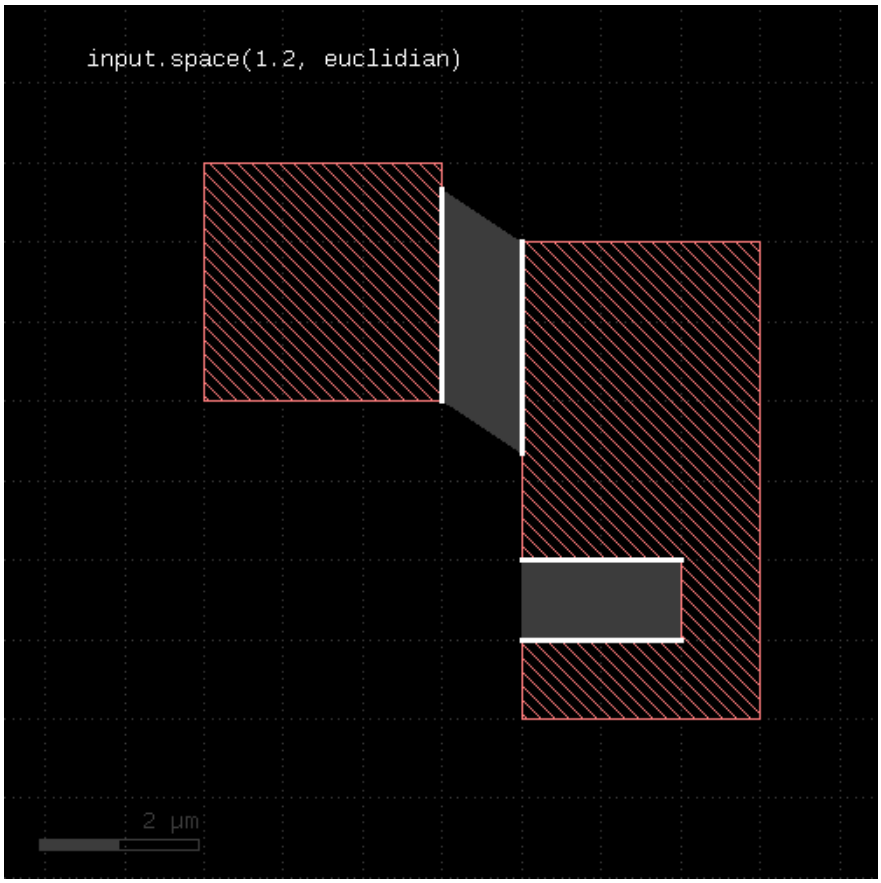
The notch method is similar, but will only report self-space violations. The isolated method will only report space violations to other polygons. separation is a two-layer space check where space is checked against polygons of another layer.

As for the other DRC methods, merged semantics applies.

Distance values can be given as floating-point values (in micron) or integer values (in database units). To explicitly specify the unit, use the unit denominators.

For the manifold options of this function see the width method description.

The following image shows the effect of the space check:



"split_covering" - Returns the results of covering and not_covering at the same time

Usage:

- `(a, b) = layer.split_covering(other [, options ])`

This method returns the polygons covering polygons from the other layer in one layer and all others in a second layer. This method is equivalent to calling covering and not_covering, but is faster than doing this in separate steps:

```
(covering, not_covering) = l1.split_covering(l2)
```

The options of this method are the same than covering.

"split_inside" - Returns the results of inside and not_inside at the same time

Usage:

- `(a, b) = layer.split_inside(other)`

This method returns the polygons or edges inside of polygons or edges from the other layer in one layer and all others in a second layer. This method is equivalent to calling inside and not_inside, but is faster than doing this in separate steps:

```
(inside, not_inside) = l1.split_inside(l2)
```

"split_interacting" - Returns the results of interacting and not_interacting at the same time

Usage:

- `(a, b) = layer.split_interacting(other [, options ])`

This method returns the polygons or edges interacting with objects from the other container in one layer and all others in a second layer. This method is equivalent to calling interacting and not_interacting, but is faster than doing this in separate steps:

```
(interacting, not_interacting) = l1.split_interacting(l2)
```

The options of this method are the same than interacting.

"split_outside" - Returns the results of outside and not_outside at the same time

Usage:

- `(a, b) = layer.split_outside(other)`

This method returns the polygons or edges outside of polygons or edges from the other layer in one layer and all others in a second layer. This method is equivalent to calling outside and not_outside, but is faster than doing this in separate steps:

```
(outside, not_outside) = l1.split_outside(l2)
```

"split_overlapping" - Returns the results of overlapping and not_overlapping at the same time

Usage:

- `(a, b) = layer.split_overlapping(other [, options ])`

This method returns the polygons overlapping polygons from the other layer in one layer and all others in a second layer. This method is equivalent to calling overlapping and not_overlapping, but is faster than doing this in separate steps:

```
(overlapping, not_overlapping) = l1.split_overlapping(l2)
```

The options of this method are the same than overlapping.

"squares" - Selects all squares from the input

Usage:

- `layer.squares`

This method is available for polygon layers. By default "merged" semantics applies, i.e. all polygons are merged before squares are selected (see clean and raw). non_squares will select all non-rectangles.

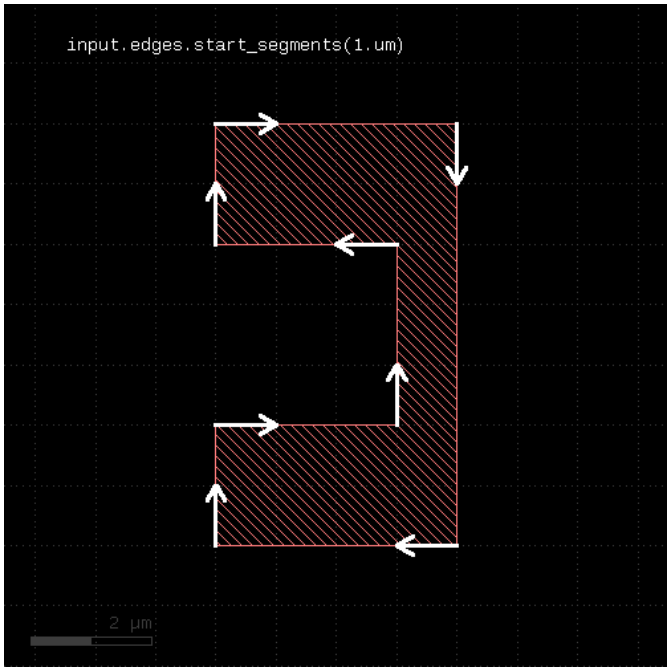
"start_segments" - Returns the part at the beginning of each edge

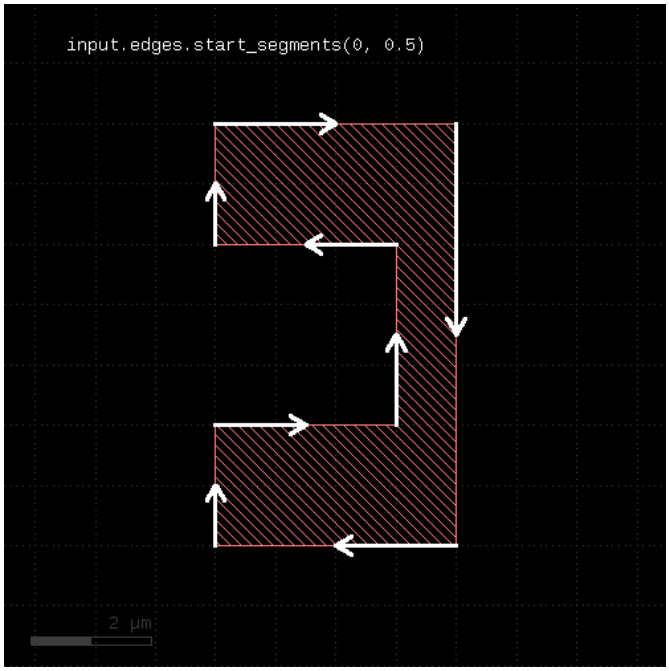
Usage:

- `layer.start_segments(length)`
- `layer.start_segments(length, fraction)`

This method will return a partial edge for each edge in the input, located and the end of the original edge. The new edges will share the start point with the original edges, but not necessarily their end points. For further details about the orientation of edges and the parameters of this method, see end_segments.

The following images show the effect of the method:





"strict" - Marks a layer for strict handling

Usage:

- `layer.strict`

If a layer is marked for strict handling, some optimizations are disabled. Specifically for boolean operations, the results will also be merged if one input is empty. For boolean operations, strict handling should be enabled for both inputs. Strict handling is disabled by default.

See non_strict about how to reset this mode.

This feature has been introduced in version 0.23.2.

"strict?" - Returns true, if strict handling is enabled for this layer

Usage:

- `layer.is_strict?`

See strict for a discussion of strict handling.

This feature has been introduced in version 0.23.2.

"texts" - Selects texts from an original layer

Usage:

- `layer.texts`
- `layer.texts(p)`
- `layer.texts([ options ])`

This method can be applied to original layers - i.e. ones that have been created with input. By default, a small box (2x2 DBU) will be produced on each selected text. By using the "as_dots" option, degenerated point-like edges will be produced.

The preferred method however is to use true text layers created with labels. In this case, without specifying "as_dots" or "as_boxes" retains the text objects as such a text filtering is applied. In contrast to this, layers generated with input cannot maintain the text nature of the selected objects and produce dots or small polygon boxes in the texts method.

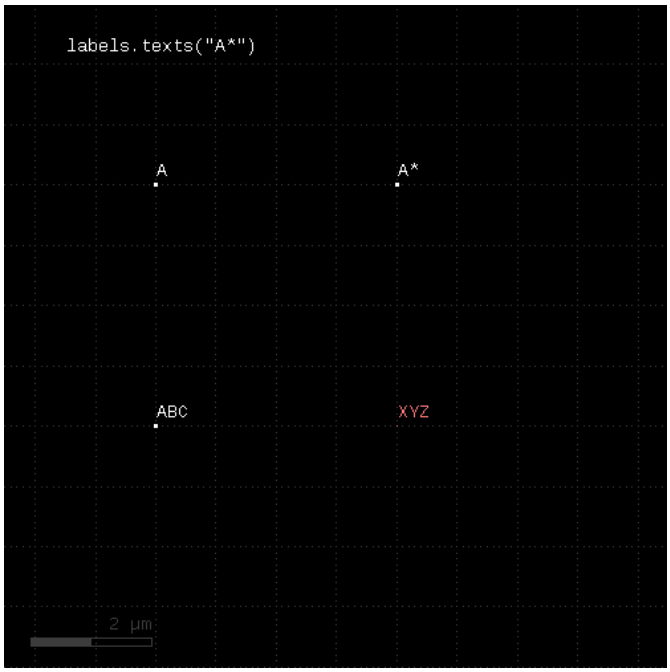
Texts can be selected either by exact match string or a pattern match with a glob-style pattern. By default, glob-style pattern are used. The options available are:

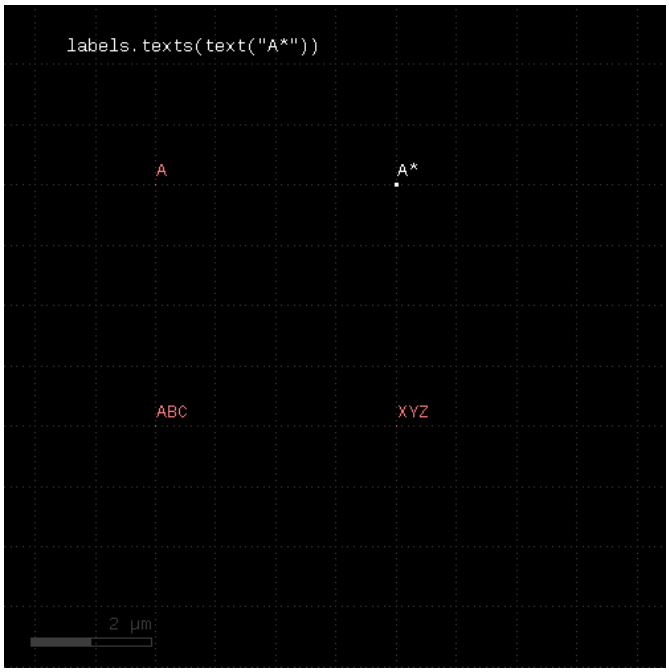
- **pattern(p)** : Use a pattern to match the string (this is the default)
- **text(s)** : Select the texts that exactly match the given string
- **as_boxes** : with this option, small boxes will be produced as markers
- **as_dots** : with this option, point-like edges will be produced instead of small boxes

Here are some examples:

```
# Selects all texts
t = labels(1, 0).texts
# Selects all texts beginning with an "A"
t = labels(1, 0).texts("A*")
t = labels(1, 0).texts(pattern("A*"))
# Selects all texts whose string is "ABC"
t = labels(1, 0).texts(text("ABC"))
```

The effect of the operation is shown in these examples:





"texts?" - Returns true, if the layer is a text collection

Usage:

- `layer.texts?`

"texts_not" - Selects texts from an original layer not matching a specific selection

Usage:

- `layer.texts_not`
- `layer.texts_not(p)`
- `layer.texts_not([ options ])`

This method can be applied to true text layers obtained with labels. In this case, without specifying "as_dots" or "as_boxes" retains the text objects as such. Only text filtering is applied.

Beside that this method acts like texts, but will select the text objects not matching the filter.

"transform" - Transforms a layer (modifies the layer)

Usage:

- `layer.transform(t)`

Like transform, but modifies the input and returns a reference to it for further processing.

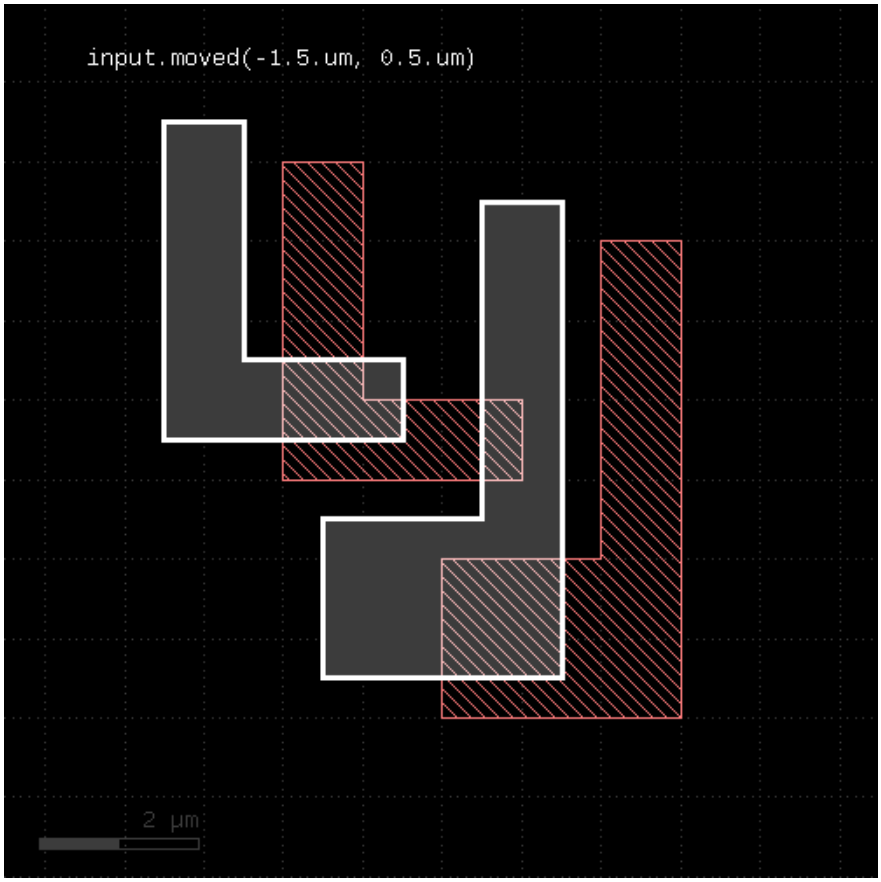
"transformed" - Transforms a layer

Usage:

- `layer.transformed(t)`

Transforms the input layer by the given transformation and returns the moved layer. The layer that this method is called upon is not modified. This is the most generic method is transform a layer. The transformation is a DCplxTrans object which describes many different kinds of affine transformations except shear and anisotropic magnification.

The following image shows the effect of the "moved" method:



"width" - A width check

Usage:

- `layer.width(value [, options])`

Note: "width" is available as an operator for the "universal DRC" function Layer#drc within the DRC framework. This variant has more options and is more intuitive to use. See width for more details.

This method performs a width check and returns a collection of edge pairs. A width check can be performed on polygon and edge layers. On edge layers, all edges are checked against all other edges. If two edges form a "back to back" relation (i.e. their inner sides face each other) and their distance is less than the specified value, an error shape is generated for that edge pair. On polygon layers, the polygons on each layer are checked for locations where their width is less than the specified value. In that case, an edge pair error shape is generated.

Options

The options available are:

- **euclidian** : perform the check using Euclidian metrics (this is the default)

- **square** : perform the check using Square metrics
- **projection** : perform the check using projection metrics
- **whole_edges** : With this option, the check will return all of the edges, even if the criterion is violated only over a part of the edge
- **angle_limit(a)** : Specifies the angle above or equal to which no check is performed. The default value is 90, which means that for edges having an angle of 90 degree or more, no check is performed. Setting this value to 45 will make the check only consider edges enclosing angles of less than 45 degree.
- **projection_limits(min, max) or projection_limits(min .. max)** : this option makes the check only consider edge pairs whose projected length on each other is more or equal than min and less than max
- **projecting (in condition)** : This specification is equivalent to "projection_limits" but is more intuitive, as "projecting" is written with a condition, like "projecting < 2.um". Available operators are: "==", "<", "<=", ">" and ">=". Double-bounded ranges are also available, like: "0.5 <= projecting < 2.0".
- **without_touching_corners** : With this option present, touching corners (aka "kissing corners") will not yield errors. The default is to produce errors in these cases.
- **without_touching_edges** : With this option present, coincident edges will not yield errors. This is a stronger version of "without_touching_corners" and makes sense only for two-layer checks or raw-mode input layers. It is listed here for completeness.
- **transparent** : Performs the check without shielding (polygon layers only)
- **shielded** : Performs the check with shielding (polygon layers only)
- **props_eq** , **props_ne** , **props_copy** : (only props_copy applies to width check) - See "Properties constraints" below.

Note that without the angle_limit, acute corners will always be reported, since two connected edges always violate the width in the corner. By adjusting the angle_limit, an acute corner check can be implemented.

Merge semantics applies to this method, i.e. disconnected polygons are merged before the width is checked unless "raw" mode is chosen.

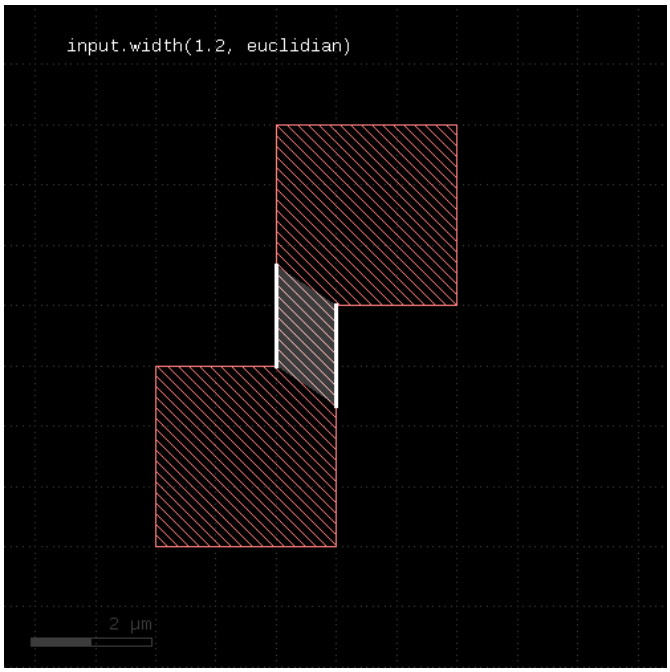
The resulting edge pairs can be converted to polygons using the polygons method.

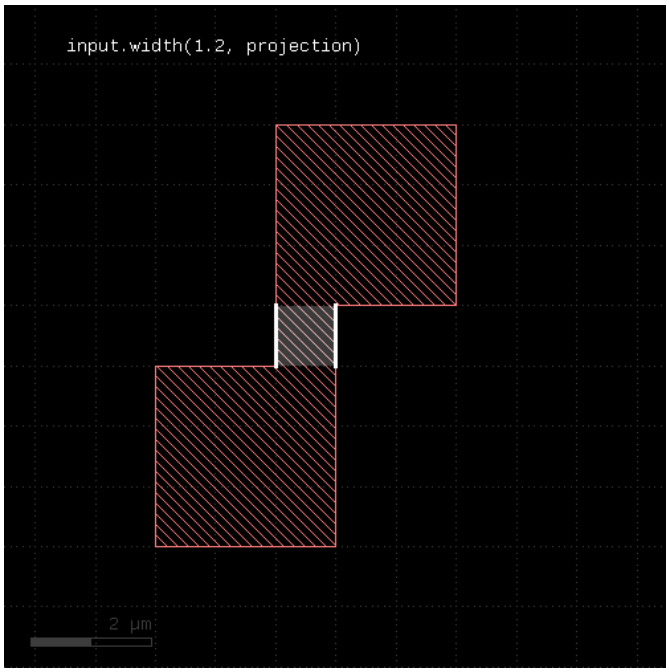
Distance values can be given as floating-point values (in micron) or integer values (in database units). To explicitly specify the unit, use the unit denominators, i.e.

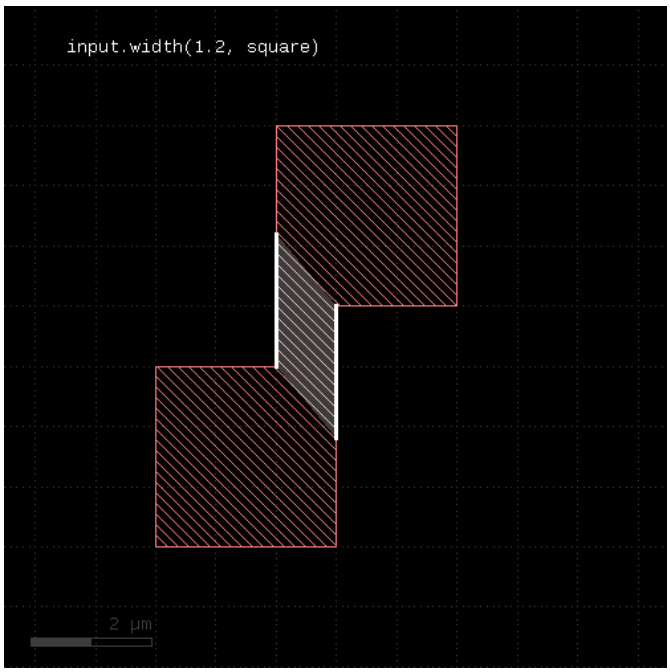
```
# width check for 1.5 micron:
markers = in.width(1.5)
# width check for 2 database units:
markers = in.width(2)
# width check for 2 micron:
markers = in.width(2.um)
# width check for 20 nanometers:
markers = in.width(20.nm)
```

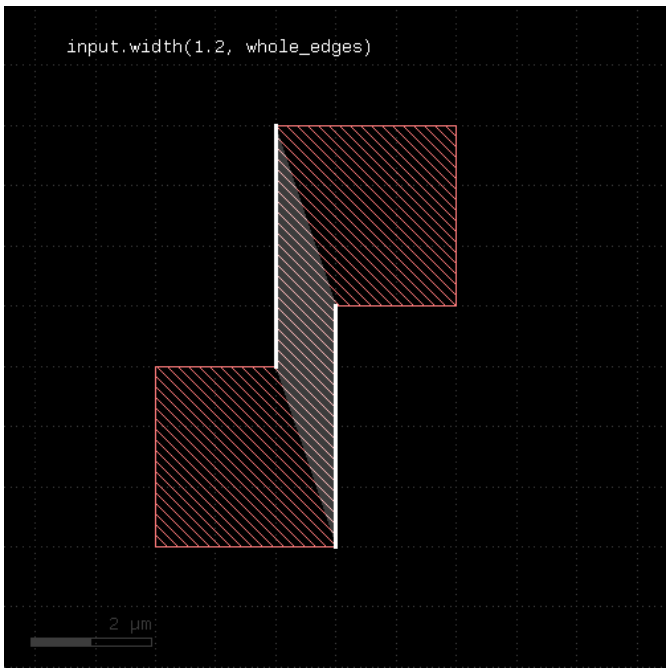
Examples

The following images show the effect of various forms of the width check:









Universal DRC function

There is an alternative notation for the check using the "universal DRC" function ("Layer#drc"). This notation is more intuitive and allows checking for widths bigger than a certain value or within a certain range. See "width" for details.

Apart from that it provides the same options than the plain width check. Follow this link for the documentation of this feature: width.

Shielding

"shielding" is a concept where an internal or external distance is measured only if the opposite edge is not blocked by other edges between. Shielded mode makes a difference if very large distances are to be checked and the minimum distance is much smaller: in this case, a large distance violation may be blocked by features located between the edges which are checked. With shielding, large distance violations are not reported in this case. Shielding is also effective at zero distance which has an adverse effect: Consider a case, where one layer A is a subset of another layer B. If you try to check the distance between features of B vs. A, you cannot use shielding, because B features which are identical to A features will shield those entirely.

Shielding is enabled by default, but can be switched off with the "transparent" option.

Properties constraints (available on intra-polygon checks such as space, sep etc.)

This feature is listed here, because this documentation is generic and used for other checks as well. props_eq and props_ne are not available on 'width' or 'notch' as these apply to intra-polygon checks - when pairs of different polygons are involved - something that 'width' or 'notch' does need.

With properties constraints, the check is performed between shapes with the same or different properties. "properties" refers to the full set of key/value pairs attached to a shape.

Property constraints are specified by adding props_eq or props_ne to the arguments. If these literals are present, only shapes with same of different properties are involved in the check. In connection with the net annotation feature this allows checking space between connected or disconnected shapes for example:

```
connect(metal1, via1)
...

# attaches net identity as properties
metal1_nets = metal1.nets

space_not_connected = metal1_nets.space(0.4.um, props_ne)
space_connected     = metal1_nets.space(0.4.um, props_eq)
```

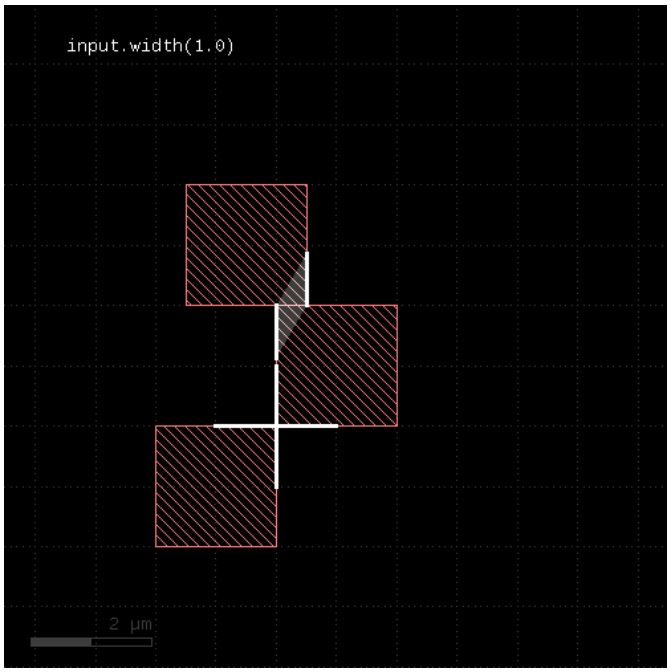
props_copy is a special properties constraint that does not alter the behaviour of the checks, but copies the primary shape's properties to the output markers. This constraint is applicable to width and notch checks too. The effect is that the original polygon's properties are copied to the error markers. props_copy can be combined with props_eq and props_ne to copy the original shape's properties to the output too:

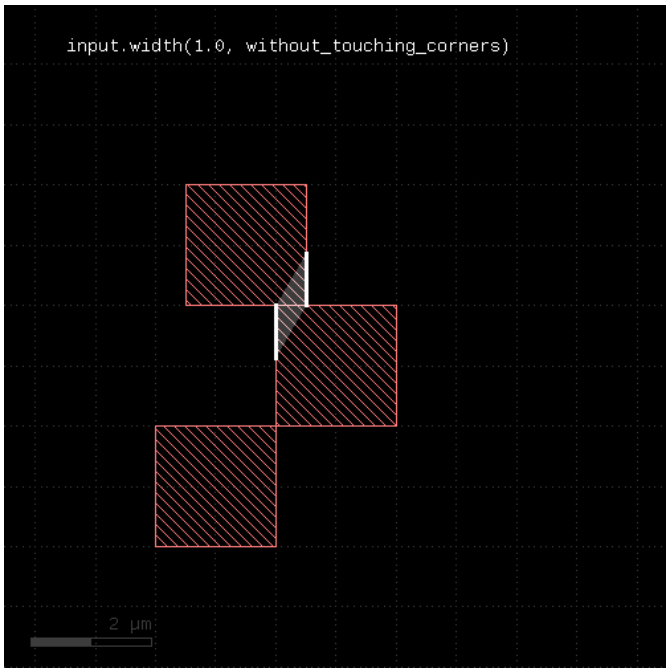
```
space_not_connected = metal1_nets.space(0.4.um, props_ne + props_copy)
space_connected     = metal1_nets.space(0.4.um, props_eq + props_copy)
```

Touching shapes

The "without_touching_corners" option will turn off errors that arise due to the "kissing corner" configuration (or "checkerboard pattern"). Formally this is a width violation across the diagonal, but when considering this configuration as disconnected boxes, no error should be reported.

The following images illustrate the effect of the "without_touching_corners" option:





"with_angle" - Selects edges by their angle

Usage:

- `layer.with_angle(min .. max)`
- `layer.with_angle(value)`
- `layer.with_angle(min, max)`
- `layer.with_angle(ortho)`
- `layer.with_angle(diagonal)`
- `layer.with_angle(diagonal_only)`
- `edge_pair_layer.with_angle(... [, both])`

When called on an edge layer, the method selects edges by their angle, measured against the horizontal axis in the mathematical sense.

For this measurement edges are considered without their direction and straight lines. A horizontal edge has an angle of zero degree. A vertical one has an angle of 90 degee. The angle range is from -90 (exclusive) to 90 degree (inclusive).

The first version of this method selects edges with a angle larger or equal to min and less than max (but not equal). The second version selects edges with exactly the given angle. The third version is identical to the first one.

When called on an edge pair layer, this method selects edge pairs with one or both edges meeting the angle criterion. In this case an additional argument is accepted which can be either "both" (plain word) to indicate that both edges have to be within the given interval. Without this argument, it is sufficient for one edge to meet the criterion.

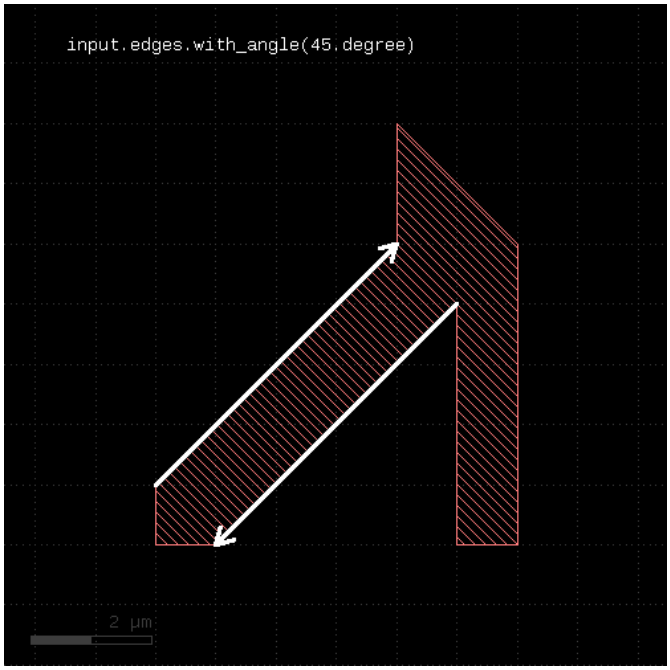
Here are examples for "with_angle" on edge pair layers:

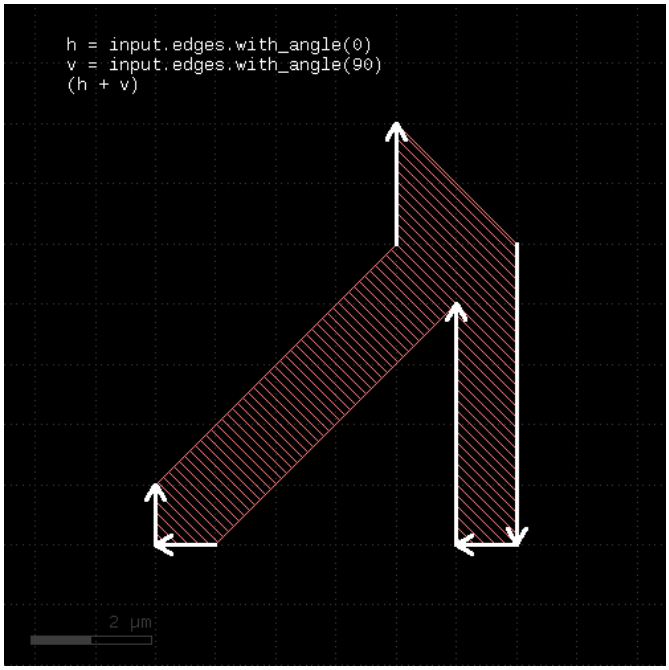
```
# at least one edge needs to be horizontal
ep1 = edge_pairs.with_angle(0)
# both edges need to vertical
```

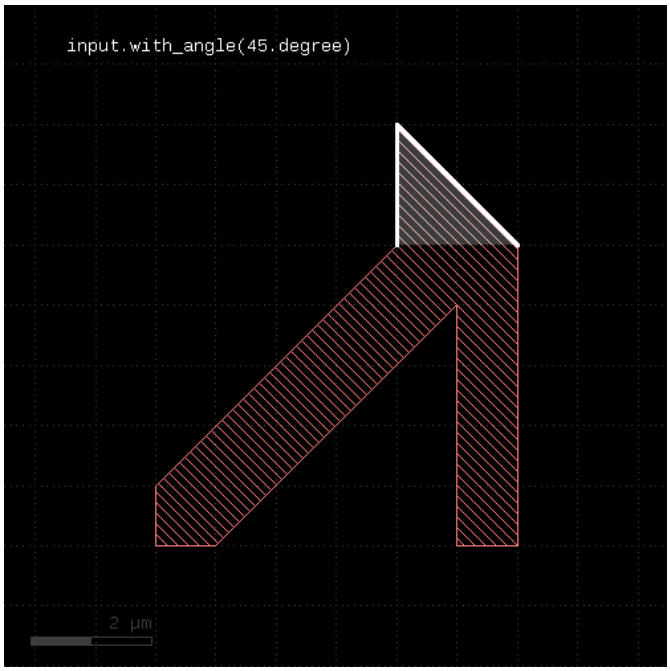
```
ep2 = edge_pairs.with_angle(90, both)
```

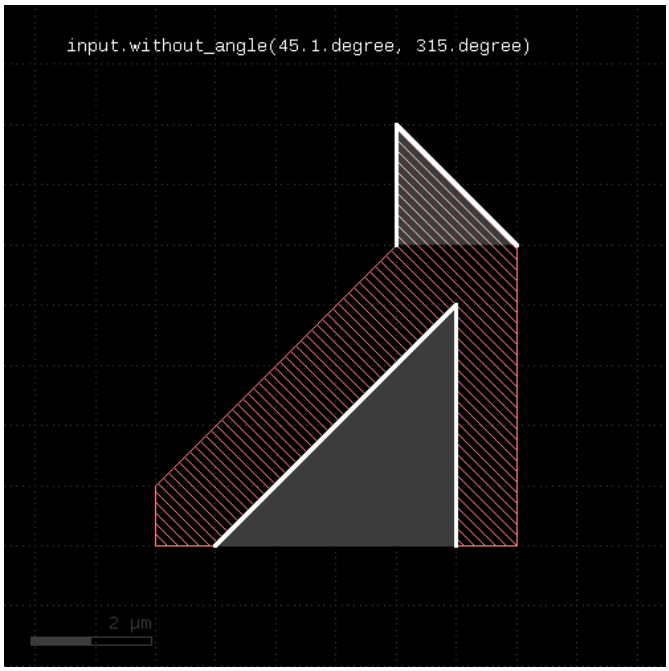
A method delivering all objects not matching the angle criterion is without_angle. Note that for edge pairs, in order to get the inverse result, you have to add or drop "both" on without_angle. This is because without_angle without both returns edge pairs where one edge does not match the criterion. The logical opposite of "one edge matches" however is "both edges do not match".

The following images demonstrate some use cases of with_angle and without_angle :









Specifying "ortho", "diagonal" or "diagonal_only" instead of the angle values will select 0 and 90 degree edges (ortho), -45 and 45 degree edges (diagonal_only) and both types (diagonal). This simplifies the implementation of selectors for manhattan or half-manhattan features:

```
ortho_edges = edges.with_angle(ortho)

# which is equivalent to, but more efficient as:
ortho_edges = edges.with_angle(0) + edges.with_angle(90)
```

Note that in former versions, with_angle could be used on polygon layers selecting corners with specific angles. This feature has been deprecated. Use corners instead.

"with_area" - Selects polygons or edge pairs by area

Usage:

- `layer.with_area(min .. max)`
- `layer.with_area(value)`
- `layer.with_area(min, max)`

The first form will select all polygons or edge pairs with an area larger or equal to min and less (but not equal to) max. The second form will select the polygons or edge pairs with exactly the given area. The third form basically is equivalent to the first form, but allows specification of nil for min or max indicating no lower or upper limit.

This method is available for polygon or edge pair layers.

"with_area_ratio" - Selects polygons by the ratio of the bounding box area vs. polygon area

Usage:

- `layer.with_area_ratio(min .. max)`
- `layer.with_area_ratio(value)`
- `layer.with_area_ratio(min, max)`

The area ratio is a measure how far a polygon is approximated by its bounding box. The value is always larger or equal to 1. Boxes have a area ratio of 1. Larger values mean more empty area inside the bounding box.

This method is available for polygon layers only.

"with_bbox_aspect_ratio" - Selects polygons by the aspect ratio of their bounding box

Usage:

- `layer.with_bbox_aspect_ratio(min .. max)`
- `layer.with_bbox_aspect_ratio(value)`
- `layer.with_bbox_aspect_ratio(min, max)`

The method selects polygons similar to with_area or with_perimeter. However, the measured value is the aspect ratio of the bounding box. It is the larger dimensions divided by the smaller one. The "thinner" the polygon, the larger the aspect ratio. A square bounding box gives an aspect ratio of 1.

This method is available for polygon layers only.

"with_bbox_height" - Selects polygons by the height of the bounding box

Usage:

- `layer.with_bbox_height(min .. max)`
- `layer.with_bbox_height(value)`
- `layer.with_bbox_height(min, max)`

The method selects polygons similar to with_area or with_perimeter. However, the measured dimension is the width of the bounding box.

This method is available for polygon layers only.

"with_bbox_max" - Selects polygons by the maximum dimension of the bounding box

Usage:

- `layer.with_bbox_max(min .. max)`
- `layer.with_bbox_max(value)`
- `layer.with_bbox_max(min, max)`

The method selects polygons similar to with_area or with_perimeter. However, the measured dimension is the maximum dimension of the bounding box. The maximum dimension is either the width or height of the bounding box, whichever is larger.

This method is available for polygon layers only.

"with_bbox_min" - Selects polygons by the minimum dimension of the bounding box

Usage:

- `layer.with_bbox_min(min .. max)`
- `layer.with_bbox_min(value)`
- `layer.with_bbox_min(min, max)`

The method selects polygons similar to with_area or with_perimeter. However, the measured dimension is the minimum dimension of the bounding box. The minimum dimension is either the width or height of the bounding box, whichever is smaller.

This method is available for polygon layers only.

"with_bbox_width" - Selects polygons by the width of the bounding box

Usage:

- `layer.with_bbox_width(min .. max)`
- `layer.with_bbox_width(value)`
- `layer.with_bbox_width(min, max)`

The method selects polygons similar to with_area or with_perimeter. However, the measured dimension is the width of the bounding box.

This method is available for polygon layers only.

"with_density" - Returns tiles whose density is within a given range

Usage:

- `layer.with_density(min_value, max_value [, options ])`
- `layer.with_density(min_value .. max_value [, options ])`

This method runs a tiled analysis over the current layout. It reports the tiles whose density is between "min_value" and "max_value". "min_value" and "max_value" are given in relative units, i.e. within the range of 0 to 1.0 corresponding to a density of 0 to 100%.

"min_value" or "max_value" can be nil or omitted in the ".." range notation. In this case, they are taken as "0" and "100%".

The tile size must be specified with the "tile_size" option:

```
# reports areas where layer 1/0 density is below 10% on 20x20 um tiles
low_density = input(1, 0).with_density(0.0 .. 0.1, tile_size(20.um))
```

Anisotropic tiles can be specified by giving two values, like "tile_size(10.um, 20.um)". The first value is the horizontal tile dimension, the second value is the vertical tile dimension.

A tile overlap can be specified using "tile_step". If the tile step is less than the tile size, the tiles will overlap. The layout window given by "tile_size" is moved in increments of the tile step:

```
# reports areas where layer 1/0 density is below 10% on 30x30 um tiles
# with a tile step of 20x20 um:
low_density = input(1, 0).with_density(0.0 .. 0.1, tile_size(30.um), tile_step(20.um))
```

For "tile_step", anisotropic values can be given as well by using two values: the first for the horizontal and the second for the vertical tile step.

Another option is "tile_origin" which specifies the location of the first tile's position. This is the lower left tile's lower left corner. If no origin is given, the tiles are centered over the area investigated.

By default, the tiles will cover the bounding box of the input layer. A separate layer can be used in addition. This way, the layout's dimensions can be derived from some drawn boundary layer. To specify a separate, additional layer included in the bounding box, use the "tile_boundary" option:

```
# reports density of layer 1/0 below 10% on 20x20 um tiles. The layout's boundary is taken from
# layer 0/0:
cell_frame = input(0, 0)
low_density = input(1, 0).with_density(0.0 .. 0.1, tile_size(20.um), tile_boundary(cell_frame))
```

Note that the layer given in "tile_boundary" adds to the input layer for computing the bounding box. The computed area is at least the area of the input layer.

Computation of the area can be skipped by explicitly giving a tile count in horizontal and vertical direction. With the "tile_origin" option this allows full control over the area covered:

```
# reports density of layer 1/0 below 10% on 20x20 um tiles in the region 0,0 .. 2000,3000
# (100 and 150 tiles of 20 um each are used in horizontal and vertical direction):
low_density = input(1, 0).with_density(0.0 .. 0.1, tile_size(20.um), tile_origin(0.0, 0.0), tile_count(100,
 150))
```

The "padding mode" indicates how the area outside the layout's bounding box is considered. There are two modes:

- **padding_zero** : the outside area is considered zero density. This is the default mode.
- **padding_ignore** : the outside area is ignored for the density computation.

Example:

```
low_density = input(1, 0).with_density(0.0 .. 0.1, tile_size(20.um), padding_ignore)
```

The complementary version of "with_density" is without_density.

"with_distance" - Selects edge pairs by the distance of the edges

Usage:

- `layer.with_distance(min .. max)`
- `layer.with_distance(value)`
- `layer.with_distance(min, max)`

The method selects edge pairs by the distance of their edges. The first version selects edge pairs with a distance larger or equal to min and less than max (but not equal). The second version selects edge pairs with exactly the given distance. The third version is similar to the first one, but allows specification of nil for min or max indicating that there is no lower or upper limit.

The distance of the edges is defined by the minimum distance of all points from the edges involved. For edge pairs generated in geometrical checks this is equivalent to the actual distance of the original edges.

This method is available for edge pair layers only.

"with_holes" - Selects all polygons with the specified number of holes

Usage:

- `layer.with_holes`
- `layer.with_holes(count)`
- `layer.with_holes(min_count, max_count)`
- `layer.with_holes(min_count .. max_count)`

This method is available for polygon layers. It will select all polygons from the input layer which have the specified number of holes. Without any argument, all polygons with holes are selected.

"with_internal_angle" - Selects edge pairs by their internal angle

Usage:

- `edge_pair_layer.with_internal_angle(min .. max)`
- `edge_pair_layer.with_internal_angle(value)`
- `edge_pair_layer.with_internal_angle(min, max)`

This method selects edge pairs by the angle enclosed by their edges. The angle is between 0 (parallel or anti-parallel edges) and 90 degree (perpendicular edges). If an interval or two values are given, the angle is checked to be within the given range.

Here are examples for "with_internal_angle" on edge pair layers:

```
# selects edge pairs with parallel edges
ep1 = edge_pairs.with_internal_angle(0)
# selects edge pairs with perpendicular edges
```

```
ep2 = edge_pairs.with_internal_angle(90)
```

"with_length" - Selects edges by their length

Usage:

- `layer.with_length(min .. max)`
- `layer.with_length(value)`
- `layer.with_length(min, max)`
- `edge_pairlayer.with_length(min, max [, both])`

The method selects edges by their length. The first version selects edges with a length larger or equal to min and less than max (but not equal). The second version selects edges with exactly the given length. The third version is similar to the first one, but allows specification of nil for min or max indicating that there is no lower or upper limit.

This method is available for edge and edge pair layers.

When called on an edge pair layer, this method will select edge pairs if one or both of the edges meet the length criterion. Use the additional argument and pass "both" (plain word) to specify that both edges need to be within the given interval. By default, it's sufficient for one edge to meet the criterion.

Here are examples for "with_length" on edge pair layers:

```
# at least one edge needs to have a length of 1.0 <= l < 2.0
ep1 = edge_pairs.with_length(1.um .. 2.um)
# both edges need to have a length of exactly 2 um
ep2 = edge_pairs.with_length(2.um, both)
```

"with_perimeter" - Selects polygons by perimeter

Usage:

- `layer.with_perimeter(min .. max)`
- `layer.with_perimeter(value)`
- `layer.with_perimeter(min, max)`

The first form will select all polygons with an perimeter larger or equal to min and less (but not equal to) max. The second form will select the polygons with exactly the given perimeter. The third form basically is equivalent to the first form, but allows specification of nil for min or max indicating no lower or upper limit.

This method is available for polygon layers only.

"with_relative_height" - Selects polygons by the ratio of the height vs. width of its bounding box

Usage:

- `layer.with_relative_height(min .. max)`
- `layer.with_relative_height(value)`
- `layer.with_relative_height(min, max)`

The relative height is a measure how tall a polygon is. Tall polygons have values larger than 1, wide polygons have a value smaller than 1. Squares have a value of 1.

Don't use this method when you can use with_area_ratio, which provides a similar measure but is isotropic.

This method is available for polygon layers only.

"without_angle" - Selects edges by the their angle

Usage:

- `layer.without_angle(min .. max)`
- `layer.without_angle(value)`
- `layer.without_angle(min, max)`
- `layer.without_angle(ortho)`
- `layer.without_angle(diagonal)`
- `layer.without_angle(diagonal_only)`
- `edge_pair_layer.without_angle(... [, both])`

The method basically is the inverse of with_angle. It selects all edges of the edge layer or corners of the polygons which do not have the given angle (second form) or whose angle is not inside the given interval (first and third form) or of the given type (other forms).

When called on edge pairs, it selects edge pairs by the angles of their edges.

A note on the "both" modifier (without_angle called on edge pairs): "both" means that both edges need to be "without_angle". For example

```
# both edges are not horizontal or:
# the edge pair is skipped if one edge is horizontal
ep = edge_pairs.without_angle(0, both)
```

See with_internal_angle and without_internal_angle to select edge pairs by the angle between the edges.

"without_area" - Selects polygons or edge pairs by area

Usage:

- `layer.without_area(min .. max)`
- `layer.without_area(value)`
- `layer.without_area(min, max)`

This method is the inverse of "with_area". It will select polygons or edge pairs without an area equal to the given one or outside the given interval.

This method is available for polygon or edge pair layers.

"without_area_ratio" - Selects polygons by the aspect ratio of their bounding box

Usage:

- `layer.without_area_ratio(min .. max)`
- `layer.without_area_ratio(value)`
- `layer.without_area_ratio(min, max)`

The method provides the opposite filter for with_area_ratio.

This method is available for polygon layers only.

"without_bbox_height" - Selects polygons by the aspect ratio of their bounding box

Usage:

- `layer.without_bbox_aspect_ratio(min .. max)`
- `layer.without_bbox_aspect_ratio(value)`
- `layer.without_bbox_aspect_ratio(min, max)`

The method provides the opposite filter for with_bbox_aspect_ratio.

This method is available for polygon layers only.

"without_bbox_max" - Selects polygons by the maximum dimension of the bounding box

Usage:

- `layer.without_bbox_max(min .. max)`
- `layer.without_bbox_max(value)`
- `layer.without_bbox_max(min, max)`

The method selects polygons similar to without_area or without_perimeter. However, the measured dimension is the maximum dimension of the bounding box. The minimum dimension is either the width or height of the bounding box, whichever is larger.

This method is available for polygon layers only.

"without_bbox_min" - Selects polygons by the minimum dimension of the bounding box

Usage:

- `layer.without_bbox_min(min .. max)`
- `layer.without_bbox_min(value)`
- `layer.without_bbox_min(min, max)`

The method selects polygons similar to without_area or without_perimeter. However, the measured dimension is the minimum dimension of the bounding box. The minimum dimension is either the width or height of the bounding box, whichever is smaller.

This method is available for polygon layers only.

"without_bbox_width" - Selects polygons by the width of the bounding box

Usage:

- `layer.without_bbox_width(min .. max)`
- `layer.without_bbox_width(value)`
- `layer.without_bbox_width(min, max)`

The method selects polygons similar to without_area or without_perimeter. However, the measured dimension is the width of the bounding box.

This method is available for polygon layers only.

"without_density" - Returns tiles whose density is not within a given range

Usage:

- `layer.without_density(min_value, max_value [, options ])`
- `layer.without_density(min_value .. max_value [, options ])`

For details about the operations and the operation see with_density. This version will return the tiles where the density is not within the given range.

"without_distance" - Selects edge pairs by the distance of the edges

Usage:

- `layer.without_distance(min .. max)`
- `layer.without_distance(value)`
- `layer.without_distance(min, max)`

The method basically is the inverse of with_distance. It selects all edge pairs of the edge pair layer which do not have the given distance (second form) or are not inside the given interval (first and third form).

This method is available for edge pair layers only.

"without_holes" - Selects all polygons with the specified number of holes

Usage:

- `layer.without_holes`
- `layer.without_holes(count)`
- `layer.without_holes(min_count, max_count)`
- `layer.without_holes(min_count .. max_count)`

This method is available for polygon layers. It will select all polygons from the input layer which do not have the specified number of holes. Without any arguments, all polygons without holes are selected.

"without_internal_angle" - Selects edge pairs by their internal angle

Usage:

- `edge_pair_layer.without_internal_angle(min .. max)`
- `edge_pair_layer.without_internal_angle(value)`
- `edge_pair_layer.without_internal_angle(min, max)`

The method basically is the inverse of with_internal_angle. It selects all edge pairs by the angle enclosed by their edges, applying the opposite criterion than with_internal_angle.

"without_length" - Selects edges by the their length

Usage:

- `layer.without_length(min .. max)`

- `layer.without_length(value)`
- `layer.without_length(min, max)`
- `edge_pairlayer.with_length(min, max [, both])`

The method basically is the inverse of with_length. It selects all edges of the edge layer which do not have the given length (second form) or are not inside the given interval (first and third form).

This method is available for edge and edge pair layers.

A note on the "both" modifier (without_length called on edge pairs): "both" means that both edges need to be "without_length". For example

```
# both edges are not exactly 1 um in length, or:
# the edge pair is skipped if one edge has a length of exactly 1 um
ep = edge_pairs.without_length(1.um, both)
```

"without_perimeter" - Selects polygons by perimeter

Usage:

- `layer.without_perimeter(min .. max)`
- `layer.without_perimeter(value)`
- `layer.without_perimeter(min, max)`

This method is the inverse of "with_perimeter". It will select polygons without a perimeter equal to the given one or outside the given interval.

This method is available for polygon layers only.

"without_relative_height" - Selects polygons by the ratio of the height vs. width

Usage:

- `layer.without_relative_height(min .. max)`
- `layer.without_relative_height(value)`
- `layer.without_relative_height(min, max)`

The method provides the opposite filter for with_relative_height.

This method is available for polygon layers only.

"xor" - Boolean XOR operation

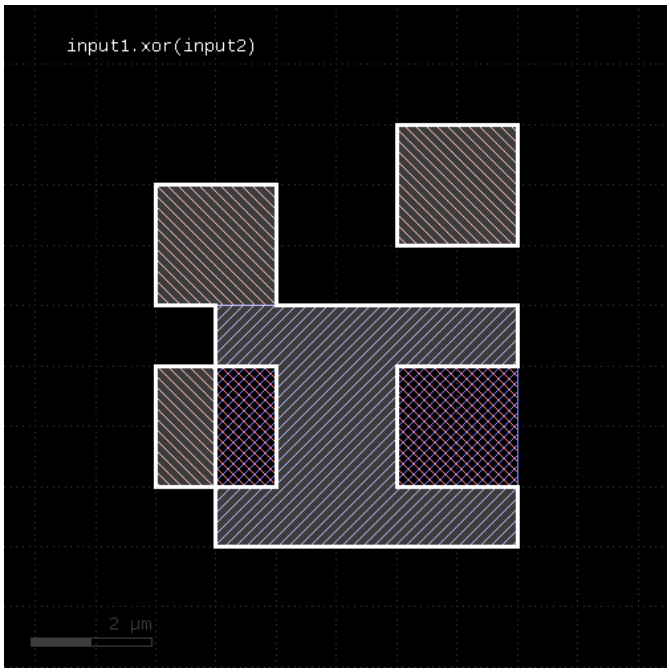
Usage:

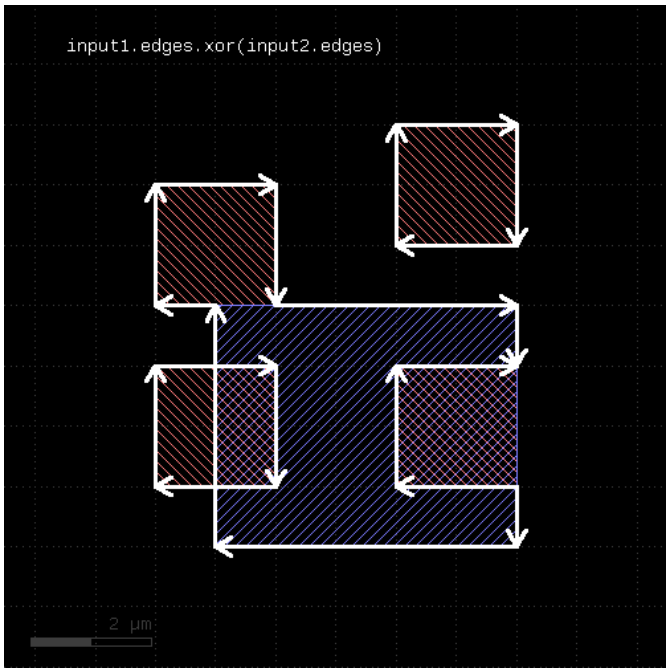
- `layer.xor(other)`

The method computes a boolean XOR between self and other. It is an alias for the "^" operator.

This method is available for polygon and edge layers.

The following images show the effect of the "xor" method on polygons and edges (input1: red, input2: blue):





"|" - Boolean OR operation

Usage:

- `self | other`

The method computes a boolean OR between self and other. A similar operation is join which will basically gives the same result but won't merge the shapes.

This method is available for polygon and edge layers. An alias is "or". See there for a description of the function.

2.20.3. DRC Reference: Netter object

The Netter object provides services related to network extraction from a layout. The relevant methods of this object are available as global functions too where they act on a default incarnation of the netter. Usually it's not required to instantiate a Netter object, but it serves as a container for this functionality.

```
# create a new Netter object:
nx = netter
nx.connect(poly, contact)
...
```

Network formation:

A basic service the Netter object provides is the formation of connected networks of conductive shapes (netting). To do so, the Netter must be given a connection specification. This happens by calling "connect" with two polygon layers. The Netter will then regard all overlaps of shapes on these layers as connections between the respective materials. Networks are the basis for netlist extraction, network geometry deduction and the antenna check.

Connections can be cleared with "clear_connections". If not, connections add atop of the already defined ones. Here is an example for the antenna check:

```
# build connction of poly+gate to metal1
connect(gate, poly)
connect(poly, contact)
connect(contact, metal1)

# runs an antenna check for metal1 with a ratio of 50
m1_antenna_errors = antenna_check(gate, metal1, 50.0)

# add connections to metal2
connect(metal1, via1)
connect(via1, metal2)

# runs an antenna check for metal2 with a ratio of 70.0
m2_antenna_errors = antenna_check(gate, metal2, 70.0)

# this will remove all connections made
clear_connections
...
```

Further functionality of the Netter object:

More methods will be added in the future to support network-related features.

"antenna_check" - Performs an antenna check

Usage:

- `antenna_check(gate, metal, ratio, [ diode_specs ... ] [, texts ])`

The antenna check is used to avoid plasma induced damage. Physically, the damage happes if during the manufacturing of a metal layer with plasma etching charge accumulates on the metal islands. On reaching a certain threshold, this charge may discarge over gate oxide attached of devices attached to such metal areas hence damaging it.

Antenna checks are performed by collecting all connected nets up to a certain metal layer and then computing the area of all metal shapes and all connected gates of a certain kind (e.g. thin and thick oxide gates). The ratio of metal area divided by the gate area must not exceed a certain threshold.

A simple antenna check is this:

```
poly = ... # poly layer
diff = ... # diffusion layer
contact = ... # contact layer
metal1 = ... # metal layer

# compute gate area
gate = poly & diff

# note that gate and poly have to be included - gate is
# a subset of poly, but forms the sensitive area
connect(gate, poly)
connect(poly, contact)
connect(contact, metal1)
errors = antenna_check(gate, metal1, 50.0)
```

Usually antenna checks apply to multiple metal layers. In this case, the connectivity needs to be extended after the first check to include the next metal layers. This can be achieved with incremental connects:

```
# provide connections up to metal1
connect(gate, poly)
connect(poly, contact)
connect(contact, metal1)
metal1_errors = antenna_check(gate, metal1, 50.0)

# now *add* connections up to metal2
connect(metal1, via1)
connect(via1, metal2)
metal2_errors = antenna_check(gate, metal2, 50.0)

... continue this scheme with further metal layers ...
```

Plasma induced damage can be rectified by including diodes which create a safe current path for discharging the metal islands. Such diodes can be identified with a recognition layer (usually the diffusion area of a certain kind). You can include such diode recognition layers in the antenna check. If a connection is detected to a diode, the respective network is skipped:

```
...
diode = ... # diode recognition layer

connect(diode, contact)
errors = antenna_check(gate, metal1, 50.0, diode)
```

You can also make diode connections decreases the sensitivity of the antenna check depending on the size of the diode. The following specification makes diode connections increase the ratio threshold by 10 per square micrometer of diode area:

```
...
diode = ... # diode recognition layer

connect(diode, contact)
# each square micrometer of diode area connected to a network
# will add 10 to the ratio:
errors = antenna_check(gate, metal1, 50.0, [ diode, 10.0 ])
```

Multiple diode specifications are allowed. Just add them to the antenna_check call.

You can include the perimeter into the area computation for the gate or metal layer or both. The physical picture is this: the side walls of the material contribute to the surface too. As the side wall area can be estimated by taking the perimeter times some material thickness, the effective area is:

```
A(eff) = A + P * t
```

Here A is the area of the polygons and P is their perimeter. t is the "thickness" in micrometer units. To specify such a condition, use the following notation:

```
errors = antenna_check(area_and_perimeter(gate, 0.5), ...)
```

"area_and_perimeter" takes the polygon layer and the thickness (0.5 micrometers in this case). This notation can be applied to both gate and metal layers. A detailed notation for the usual, area-only case is available as well for completeness:

```
errors = antenna_check(area_only(gate), ...)

# this is equivalent to a zero thickness:
errors = antenna_check(area_and_perimeter(gate, 0.0), ...)
# or the standard case:
errors = antenna_check(gate, ...)
```

Finally there is also "perimeter_only". When using this specification with a thickness value, the area is computed from the perimeter alone:

```
A(eff) = P * t
```

```
errors = antenna_check(perimeter_only(gate, 0.5), ...)
```

The error shapes produced by the antenna check are copies of the metal shapes on the metal layers of each network violating the antenna rule.

You can specify a text layer (use "labels" to create one). It will receive error labels describing the measured values and computation parameters for debugging the layout. This option has been introduced in version 0.27.11.

"clear_connections" - Clears all connections stored so far

Usage:

- `clear_connections`

See connect for more details.

"connect" - Specifies a connection between two layers

Usage:

- `connect(a, b)`

a and b must be polygon or text layers. After calling this function, the Netter regards all overlapping or touching shapes on these layers to form an electrical connection between the materials formed by these layers. This also implies intra-layer connections: shapes on these layers touching or overlapping other shapes on these layers will form bigger, electrically connected areas.

Texts will be used to assign net names to the nets. The preferred method is to use labels to create a text layer from a design layer. When using input, text labels are carried implicitly with the polygons but at the cost of small dummy shapes (2x2 DBU marker polygons) and limited functionality.

Multiple connect calls must be made to form larger connectivity stacks across multiple layers. Such stacks may include forks and joins.

Connections are accumulated. The connections defined so far can be cleared with clear_connections.

"connect_explicit" - Specifies a list of net names for nets to connect ("must connect" nets)

Usage:

- `connect_explicit(net_names)`
- `connect_explicit(cell_pattern, net_names)`

Use this method to explicitly connect nets even if there is no physical connection. The concept is similar to implicit connection (see connect_implicit). The method gets a list of nets which are connected virtually, even if there is no physical connection. The first version applies this scheme to all cells, the second version to cells matching the cell name pattern. The cell name pattern follows the usual glob style form (e.g. "A*" applies the connection in all cells whose name starts with "A").

This method is useful to establish a logical connection which is made later up on the next level of hierarchy. For example, a standard cell my not contain substrate or well taps as these may be made by tap or spare cells. Logically however, the cell only has one power or ground pin for the devices and substrate or well. In order to match both representations - for example for the purpose of LVS - the dual power or ground pins have to be connected. Assuming that there is a global net "BULK" for the substrate and a net "VSS" for the sources of the NMOS devices, the following statement will create this connection for all cell names beginning with "INV":

```
connect_global(bulk, "BULK")
...
connect_explicit("INV*", [ "BULK", "VSS" ])
```

The resulting net and pin will carry a name made from the combination of the connected nets. In this case it will be "BULK,VSS".

The virtual connection in general bears the risk of missing a physical connection. The "connect_explicit" feature therefore checks whether the connection is made physically on the next hierarchy level ("must connect" nets), except for top-level cells for which it is assumed that this connection is made later. A warning is raised instead for top level cells.

Explicit connections also imply implicit connections between different parts of one of the nets. In the example before, "VSS" pieces without a physical connection will also be connected.

The explicit connections are applied on the next net extraction and cleared on "clear_connections".

"connect_global" - Connects a layer with a global net

Usage:

- `connect_global(l, name)`

Connects the shapes from the given layer l to a global net with the given name. Global nets are common to all cells. Global nets automatically connect to parent cells throughs implied pins. An example is the substrate (bulk) net which connects to shapes belonging to tie-down diodes. "l" can be a polygon or text layer.

"connect_implicit" - Specifies a search pattern for implicit net connections ("must connect" nets)

Usage:

- `connect_implicit(label_pattern)`
- `connect_implicit(cell_pattern, label_pattern)`

This method specifies a net name search pattern, either for all cells or for certain cells, given by a name search pattern. Search pattern follow the usual glob form (e.g. "A*" for all cells or nets with names starting with "A").

Then, for nets matching the net name pattern and for which there is more than one subnet, the subnets are connected. "Subnets" are physically disconnected parts of a net which carry the same name.

This feature is useful for example for power nets which are complete in a cell, but are supposed to be connected upwards in the hierarchy ("must connect" nets). Physically there are multiple nets, logically - and specifically in the schematic for the purpose of LVS - there is only one net. "connect_implicit" now creates a virtual, combined physical net that matches the logical net.

This is general bears the risk of missing a physical connection. The "connect_implicit" feature therefore checks whether the connection is made physically on the next hierarchy level, except for top-level cells for which it is assumed that this connection is made later. A warning is raised instead for top level cells.

The implicit connections are applied on the next net extraction and cleared on "clear_connections". Another feature is connect_explicit which allows connecting differently named subnets in a similar fashion.

"device_scaling" - Specifies a dimension scale factor for the geometrical device properties

Usage:

- `device_scaling(factor)`

Specifying a factor of 2 will make all devices being extracted as if the geometries were two times larger. This feature is useful when the drawn layout does not correspond to the physical dimensions.

"extract_devices" - Extracts devices based on the given extractor class, name and device layer selection

Usage:

- `extract_devices(extractor, layer_hash)`
- `extract_devices(extractor_class, name, layer_hash)`

Runs the device extraction for given device extractor class. In the first form, the extractor object is given. In the second form, the extractor's class object and the new extractor's name is given.

The device extractor is either an instance of one of the predefined extractor classes (e.g. obtained from the utility methods such as mos4) or a custom class. It provides the algorithms for deriving the device parameters from the device geometry. It needs several device recognition layers which are passed in the layer hash.

Predefined device extractors are:

- mos3 - A three-terminal MOS transistor
- mos4 - A four-terminal MOS transistor
- dmos3 - A three-terminal MOS asymmetric transistor
- dmos4 - A four-terminal MOS asymmetric transistor
- bjt3 - A three-terminal bipolar transistor
- bjt4 - A four-terminal bipolar transistor
- diode - A planar diode
- resistor - A resistor
- resistor_with_bulk - A resistor with a separate bulk terminal
- capacitor - A capacitor
- capacitor_with_bulk - A capacitor with a separate bulk terminal

Each device class (e.g. n-MOS/p-MOS or high Vt/low Vt) needs its own instance of device extractor. The device extractor beside the algorithm and specific extraction settings defines the name of the device to be built.

The layer hash is a map of device type specific functional names (key) and polygon layers (value). Here is an example:

```
deep

nwell   = input(1, 0)
active  = input(2, 0)
poly    = input(3, 0)
bulk    = make_layer   # renders an empty layer used for putting the terminals on

nactive = active - nwell  # active area of NMOS
nsd     = nactive - poly  # source/drain area
gate    = nactive & poly  # gate area

extract_devices(mos4("NMOS4"), { :SD => nsd, :G => gate, :P => poly, :W => bulk })
```

The return value of this method will be the device class of the devices generated in the extraction step (see DeviceClass).

"ignore_extraction_errors" - Specifies whether to ignore extraction errors

Usage:

- `ignore_extraction_errors(value)`

With this value set to false (the default), "extract_netlist" will raise an exception upon extraction errors. Otherwise, extraction errors will be logged but no error is raised.

"l2n_data" - Gets the internal LayoutToNetlist object

Usage:

- `l2n_data`

The LayoutToNetlist object provides access to the internal details of the netter object.

"netlist" - Gets the extracted netlist or triggers extraction if not done yet

Usage:

- `netlist`

If no extraction has been performed yet, this method will start the layout analysis. Hence, all connect, connect_global and connect_implicit calls must have been made before this method is used. Further connect statements will clear the netlist and re-extract it again.

"soft_connect" - Specifies a soft connection between two layers

Usage:

- `soft_connect(a, b)`

a and b must be polygon or text layers. After calling this function, the Netter considers shapes from layer a and b connected in "soft mode". Typically, b is a high-ohmic layer such as diffusion, implant for substate material, also called the "lower" layer.

A soft connection between shapes from layer a and b forms a directional connection like an ideal diode: current can flow down, but now up (not meant in the physical sense, this is a concept).

Hence, two nets are disconnected, if they both connect to the same lower layer, but do not have a connection between them.

The netlist extractor will use this scheme to identify nets that are connected only via such a high-ohmic region. Such a case is typically bad for the functionality of a device and reported as an error. Once, the check has been made and no error is found, soft-connected nets are joined the same way than hard connections are made.

Beside this, soft connections follow the same rules than hard connections (see connect).

"soft_connect_global" - Soft-connects a layer with a global net

Usage:

- `soft-connect_global(l, name)`

Connects the shapes from the given layer l to a global net with the given name in "soft mode".

See connect_global for details about the concepts of global nets. See soft_connect for details about the concept of soft connections. In global net soft connections, the global net (typically a substrate) is always the "lower" layer.

"top_level" - Specifies top level mode

Usage:

- `top_level(value)`

With this value set to false (the default), it is assumed that the circuit is not used as a top level chip circuit. In that case, for example must-connect nets which are not connected are reported as as warnings. If top level mode is set to true, such disconnected nets are reported as errors as this indicates a missing physical connection.

2.20.4. DRC Reference: Source Object

The layer object represents a collection of polygons, edges or edge pairs. A source specifies where to take layout from. That includes the actual layout, the top cell and options such as clip/query boxes, cell filters etc.

"cell" - Specifies input from a specific cell

Usage:

- `source.cell(name)`

This method will create a new source that delivers shapes from the specified cell.

"cell_name" - Returns the name of the currently selected cell

Usage:

- `cell_name`

"cell_obj" - Returns the Cell object of the currently selected cell

Usage:

- `cell_obj`

"clip" - Specifies clipped input

Usage:

- `source.clip(box)`
- `source.clip(p1, p2)`
- `source.clip(l, b, r, t)`

Creates a source which represents a rectangular part of the original input. Three ways are provided to specify the rectangular region: a single DBox object (micron units), two DPoint objects (lower/left and upper/right coordinate in micron units) or four coordinates: left, bottom, right and top coordinate.

This method will create a new source which delivers the shapes from that region clipped to the rectangle. A method doing the same but without clipping is touching or overlapping.

"edge_pairs" - Gets the edge pairs from an input layer

Usage:

- `source.edge_pairs`
- `source.edge_pairs(layer)`
- `source.edge_pairs(layer, datatype)`
- `source.edge_pairs(layer_into)`
- `source.edge_pairs(filter, ...)`

Creates a layer with the edge_pairs from the given layer of the source. Edge pairs are not supported by layout formats so far. So except if the source is a custom-built layout object, this method has little use. It is provided for future extensions which may include edge pairs in file streams.

This method is identical to input with respect to the options supported.

Use the global version of "edge_pairs" without a source object to address the default source.

"edge_pairs" without any arguments will create a new, empty original layer.

This method has been introduced in version 0.27.

"edges" - Gets the edge shapes (or shapes that can be converted edges) from an input layer

Usage:

- `source.edges`
- `source.edges(layer)`
- `source.edges(layer, datatype)`
- `source.edges(layer_into)`
- `source.edges(filter, ...)`

Creates a layer with the edges from the given layer of the source. Edge layers are formed from shapes by decomposing the shapes into edges: polygons for example are decomposed into their outline edges. Some file formats support egdes as native objects.

This method is identical to input with respect to the options supported.

Use the global version of "edges" without a source object to address the default source.

"edges" without any arguments will create a new, empty original layer.

This method has been introduced in version 0.27.

"extent" - Returns a layer with the bounding box of the selected layout or cells

Usage:

- `source.extent`
- `source.extent(cell_filter)`

Without an argument, the extent method returns a layer with the bounding box of the top cell. With a cell filter argument, the method returns a layer with the bounding boxes of the selected cells. The cell filter is a glob pattern.

The extent function is useful to invert a layer:

```
inverse_1 = extent.sized(100.0) - input(1, 0)
```

The following example returns the bounding boxes of all cells whose names start with "A":

```
a_cells = extent("A*")
```

"global_transform" - Gets or sets a global transformation

Usage:

- `global_transform`
- `global_transform([ transformations ])`

This method returns a new source representing the transformed layout. It is provided in the spritit of Source#clip and similar methods.

The transformation is either given as a DTrans, DVector or DCplxTrans object or as one of the following specifications:

- "shift(x, y)": shifts the input layout horizontally by x and vertically by y micrometers
- "rotate(a)": rotates the input layout by a degree counter-clockwise
- "magnify(m)": magnifies the input layout by the factor m (NOTE: using fractional scale factors may result in small gaps due to grid snapping)
- "mirror_x": mirrors the input layout at the x axis
- "mirror_y": mirrors the input layout at the y axis

Multiple transformation specs can be given. In that case the transformations are applied right to left. Using "global_transform" will reset any global transformation present already. Without an argument, the global transformation is reset.

The following example rotates the layout by 90 degree at the origin (0, 0) and then shifts it up by 100 micrometers:

```
source.global_transform(shift(0, 100.um), rotate(90.0))
```

"input" - Specifies input from a source

Usage:

- `source.input`
- `source.input(layer)`
- `source.input(layer, datatype)`
- `source.input(layer_into)`
- `source.input(filter, ...)`
- `source.input(props_spec, ...)`

Creates a layer with the shapes from the given layer of the source. The layer can be specified by layer and optionally datatype, by a LayerInfo object or by a sequence of filters. Filters are expressions describing ranges of layers and/or datatype numbers or layer names. Multiple filters can be given and all layers matching at least one of these filter expressions are joined to render the input layer for the DRC engine.

Some filter expressions are:

- `1/0-255` : Datatypes 0 to 255 for layer 1
- `1-10` : Layers 1 to 10, datatype 0
- `METAL` : A layer named "METAL"
- `METAL (17/0)` : A layer named "METAL" or layer 17, datatype 0 (for GDS, which does not have names)

Layers created with "input" may contain both texts (labels) and polygons. There is a subtle difference between flat and deep mode: in flat mode, texts are not visible in polygon operations. In deep mode, texts appear as small 2x2 DBU rectangles. In flat mode, some operations such as clipping are not fully supported for texts. Also, texts will vanish in most polygon operations such as booleans etc.

Texts can later be selected on the layer returned by "input" with the Layer#texts method.

If you don't want to see texts, use polygons to create an input layer with polygon data only. If you only want to see texts, use labels to create an input layer with texts only.

labels also produces a true "text layer" which contains text objects. A variety of operations is available for these objects, such as boolean "and" and "not" with a polygon layer. True text layers should be preferred over mixed polygon/text layers if text object processing is required.

"input" without any arguments will create a new, empty original layer.

If you want to use user properties - for example with properties constraints in DRC checks - you need to enable properties on input:

```
input1_with_props = input(1, 0, enable_props)
```

You can also filter or map property keys, similar to the functions available on layers (DRCLayer#map_props, DRCLayer#select_props). For example to select property values with key 17 (numerical) only, use:

```
input1_with_props = input(1, 0, select_props(17))
```

Use the global version of "input" without a source object to address the default source.

"labels" - Gets the labels (texts) from an input layer

Usage:

- `source.labels`
- `source.labels(layer)`
- `source.labels(layer, datatype)`
- `source.labels(layer_into)`
- `source.labels(filter, ...)`

Creates a true text layer with the labels from the given layer of the source.

This method is identical to input, but takes only texts from the given input layer. Starting with version 0.27, the result is no longer a polygon layer that tries to provide text support but a layer type which is provided for carrying text objects explicitly.

"labels" without any arguments will create a new, empty original layer.

Use the global version of "labels" without a source object to address the default source.

"layers" - Gets the layers the source contains

Usage:

- `source.layers`

Delivers a list of LayerInfo objects representing the layers inside the source.

One application is to read all layers from a source. In the following example, the "and" operation is used to perform a clip with the given rectangle. Note that this solution is not efficient - it's provided as an example only:

```
output_cell("Clipped")

clip_box = polygon_layer
clip_box.insert(box(0.um, -4.um, 4.um, 0.um))
```

```
layers.each { |l| (input(l) & clip_box).output(l) }
```

"layout" - Returns the Layout object associated with this source

Usage:

- `layout`

"make_layer" - Creates an empty polygon layer based on the hierarchy of the layout

Usage:

- `make_layer`

This method delivers a new empty original layer. It is provided to keep old code working. Use "input" without arguments instead.

"overlapping" - Specifies input selected from a region in overlapping mode

Usage:

- `source.overlapping(...)`

Like clip, this method will create a new source delivering shapes from a specified rectangular region. In contrast to clip, all shapes overlapping the region with their bounding boxes are delivered as a whole and are not clipped. Hence shapes may extent beyond the limits of the specified rectangle.

touching is a similar method which delivers shapes touching the search region with their bounding box (without the requirement to overlap)

"path" - Gets the path of the corresponding layout file or nil if there is no path

Usage:

- `path`

"polygons" - Gets the polygon shapes (or shapes that can be converted polygons) from an input layer

Usage:

- `source.polygons`
- `source.polygons(layer)`
- `source.polygons(layer, datatype)`
- `source.polygons(layer_into)`
- `source.polygons(filter, ...)`

Creates a layer with the polygon shapes from the given layer of the source. With "polygon shapes" we mean all kind of shapes that can be converted to polygons. Those are boxes, paths and real polygons.

This method is identical to input with respect to the options supported.

"polygons" without any arguments will create a new, empty original layer.

Use the global version of "polygons" without a source object to address the default source.

"select" - Adds cell name expressions to the cell filters

Usage:

- `new_source = source.select(filter1, filter2, ...)`

This method will construct a new source object with the given cell filters applied. Note that there is a global version of "select" which does not create a new source, but acts on the default source.

Cell filters will enable or disable cells plus their subtree. Cells can be switched on and off, which makes the hierarchy traversal stop or begin delivering shapes at the given cell. The arguments of the select method form a sequence of enabling or disabling instructions using cell name pattern in the glob notation ("*" as the wildcard, like shell). Disabling instructions start with a "-", enabling instructions with a "+" or no specification.

The following options are available:

- + *name_filter* : Cells matching the name filter will be enabled
- *name_filter* : Same as "+name_filter"
- - *name_filter* : Cells matching the name filter will be disabled

To disable the TOP cell but enabled a hypothetical cell B below the top cell, use that code:

```
source_with_selection = source.select("-TOP", "+B")
l1 = source_with_selection.input(1, 0)
...
```

Please note that the sample above will deliver the children of "B" because there is nothing said about how to proceed with cells other than "TOP" or "B". Conceptually, the instantiation path of a cell will be matched against the different filters in the order they are given. A matching negative expression will disable the cell, a matching positive expression will enable the cell. Hence, every cell that has a "B" in the instantiation path is enabled.

The following code will just select "B" without its children, because in the first "-*" selection, all cells including the children of "B" are disabled:

```
source_with_selection = source.select("-*", "+B")
l1 = source_with_selection.input(1, 0)
...
```

The short form "-" will disable the top cell. This code is identical to the first example and will start with a disabled top cell regardless of its name:

```
source_with_selection = source.select("-", "+B")
l1 = source_with_selection.input(1, 0)
...
```

"touching" - Specifies input selected from a region in touching mode

Usage:

- `source.touching(box)`
- `source.touching(p1, p2)`

- `source.touching(l, b, r, t)`

Like clip, this method will create a new source delivering shapes from a specified rectangular region. In contrast to clip, all shapes touching the region with their bounding boxes are delivered as a whole and are not clipped. Hence shapes may extent beyond the limits of the specified rectangle.

overlapping is a similar method which delivers shapes overlapping the search region with their bounding box (and not just touching)

2.20.5. DRC Reference: Global Functions

"angle" - In universal DRC context: selects edges based on their orientation

Usage:

- `angle (in condition)`

"angle" represents the edge orientation filter on the primary shape edges in DRC expressions (see Layer#drc and DRC#angle for more details). In this context, the operation acts similar to Layer#with_angle.

"antenna_check" - Performs an antenna check

Usage:

- `antenna_check(gate, metal, ratio, [ diode_specs ... ])`

See Netter#antenna_check for a description of that function.

"area" - Computes the total area or in universal DRC context: selects the primary shape if the area is meeting the condition

Usage:

- `area (in condition)`
- `area(layer)`

This function can be used with a layer argument. In this case it is equivalent to "layer.area" (see Layer#area) and returns the total area of the polygons in the layer.

Without a layer argument, "area" represents an area filter for primary shapes in global expressions (see Layer#drc and DRC#area for more details).

"area_ratio" - Selects primary shapes based on the ratio of bounding box and polygon area

Usage:

- `area_ratio (in condition)`

See Layer#drc, area_ratio and DRC#area_ratio for more details.

"bbox_aspect_ratio" - Selects primary shapes based on the aspect ratio of their bounding boxes

Usage:

- `bbox_aspect_ratio (in condition)`

See Layer#drc, bbox_aspect_ratio and DRC#bbox_aspect_ratio for more details.

"bbox_height" - Selects primary shapes based on their bounding box height

Usage:

- `bbox_height (in condition)`

This method creates a universal DRC expression (see Layer#drc) to select primary shapes whose bounding box height satisfies the condition. Conditions may be written as arithmetic comparisons against numeric values. For example, "bbox_height < 2.0" will select all primary shapes whose bounding box height is less than 2 micrometers. See Layer#drc for more details about comparison specs. Plain "bbox_min" is equivalent to "primary.bbox_min" - i.e. it is used on the primary shape. Also see DRC#bbox_min.

"bbox_max" - Selects primary shapes based on their bounding box height or width, whichever is larger

Usage:

- `bbox_max (in condition)`

See Layer#drc, bbox_max and DRC#bbox_max for more details.

"bbox_min" - Selects primary shapes based on their bounding box height or width, whichever is smaller

Usage:

- `bbox_max (in condition)`

See Layer#drc, bbox_min and DRC#bbox_min for more details.

"bbox_width" - Selects primary shapes based on their bounding box width

Usage:

- `bbox_max (in condition)`

See Layer#drc, bbox_height and DRC#bbox_height for more details.

"bjt3" - Supplies the BJT3 transistor extractor class

Usage:

- `bjt3(name)`
- `bjt3(name, class)`

Use this class with extract_devices to specify extraction of a bipolar junction transistor

See DeviceExtractorBJT3Transistor for more details about this extractor.

"bjt4" - Supplies the BJT4 transistor extractor class

Usage:

- `bjt4(name)`
- `bjt4(name, class)`

Use this class with extract_devices to specify extraction of a bipolar junction transistor with a substrate terminal

See DeviceExtractorBJT4Transistor for more details about this extractor.

"box" - Creates a box object

Usage:

- `box(...)`

This function creates a box object. The arguments are the same than for the DBox constructors.

"capacitor" - Supplies the capacitor extractor class

Usage:

- `capacitor(name, area_cap)`
- `capacitor(name, area_cap, class)`

Use this class with extract_devices to specify extraction of a capacitor. The area_cap argument is the capacitance in Farad per square micrometer.

See DeviceExtractorCapacitor for more details about this extractor.

"capacitor_with_bulk" - Supplies the capacitor extractor class that includes a bulk terminal

Usage:

- `capacitor_with_bulk(name, area_cap)`
- `capacitor_with_bulk(name, area_cap, class)`

Use this class with extract_devices to specify extraction of a capacitor with a bulk terminal. The area_cap argument is the capacitance in Farad per square micrometer.

See DeviceExtractorCapacitorWithBulk for more details about this extractor.

"cell" - Selects a cell for input on the default source

Usage:

- `cell(args)`

See Source#cell for a description of that function. In addition to the functionality described there, the global function will also send the output to the specified cell.

The following code will select cell "MACRO" from the input layout:

```
cell("MACRO")
# shapes now will be taken from cell "MACRO"
l1 = input(1, 0)
```

"cheat" - Hierarchy cheats

Usage:

- `cheat(args) { block }`

Hierarchy cheats can be used in deep mode to shortcut hierarchy evaluation for certain cells and consider their local configuration only. Cheats are useful for example when dealing with memory arrays. Often such arrays are build from unit cells and those often overlap with their neighbors. Now, if the hierarchical engine encounters such a situation, it will first analyse all these interactions (which can be expensive) and then it may come to the conclusion that boundary instances need to be handled differently than inside instances. This in turn might lead to propagation of shapes and in an LVS context to device externalisation: because some devices might have different parameters for boundary cells than for inside cells, the device instances can no longer be kept inside the unit cell. Specifically for memory arrays, this is not desired as eventually this leads to flattening of the whole array.

The solution is to cheat: provided the unit cell is fully fledged and neighbors do not disturb the unit cell's configuration in critical ways, the unit cell can be treated as being isolated and results are put together in the usual way.

Cheats can be applied on layout operations - specifically booleans - and device extraction operations. Cheats are only effective in deep mode.

For booleans, a cheat means that the cheating cell's boolean results are computed locally and are combined afterwards. A cheat is introduced this way:

```
deep

l1 = input(1, 0)
l2 = input(2, 0)

# usual booleans
l1and2 = l1 & l2

# will compute "UNIT_CELL" isolated and everything else in normal hierarchical mode:
l1minus2 = cheat("UNIT_CELL") { l1 - l2 }
```

The cheat block can also be wrapped in do .. end statements and can return multiple layer objects:

```
deep

l1 = input(1, 0)
l2 = input(2, 0)

# computes both AND and NOT of l1 and l2 with cheating for "UNIT_CELL"
l1and2, l1minus2 = cheat("UNIT_CELL") do
[ l1 & l2, l1 - l2 ]
end
```

(Technically, the cheat code block is a Ruby Proc and cannot create variables outside its scope. Hence the results of this code block have to be passed through the "cheat" method).

To apply cheats for device extraction, use the following scheme:

```
deep

poly = input(1, 0)
active = input(2, 0)

sd = active - poly
gate = active & poly

# device extraction with cheating for "UNIT_CELL":
cheat("UNIT_CELL") do
extract_devices(mos3("NMOS"), { "SD" => sd, "G" => gate, "tS" => sd, "tD" => sd, "tG" => poly }
end
```

The argument to the cheat method is a list of cell name pattern (glob-style pattern). For example:

```
cheat("UNIT_CELL*") { ... }
cheat("UNIT_CELL1", "UNIT_CELL2") { ... }
cheat("UNIT_CELL{1,2}") { ... }
```

For LVS applications, it's usually sufficient to cheat in the device extraction step. Cheats have been introduced in version 0.26.1.

"clear_connections" - Clears all connections stored so far

Usage:

- `clear_connections`

See Netter#clear_connections for a description of that function.

"clip" - Specifies clipped input on the default source

Usage:

- `clip(args)`

See Source#clip for a description of that function.

The following code will select shapes within a 500x600 micron rectangle (lower left corner at 0,0) from the input layout. The shapes will be clipped to that rectangle:

```
clip(0.mm, 0.mm, 0.5.mm, 0.6.mm)
# shapes now will be taken from the given rectangle and clipped to it
l1 = input(1, 0)
```

To remove the clip condition, call "clip" without any arguments.

"connect" - Specifies a connection between two layers

Usage:

- `connect(a, b)`

See Netter#connect for a description of that function.

"connect_explicit" - Specifies explicit net connections

Usage:

- `connect_explicit(net_names)`
- `connect_explicit(cell_pattern, net_names)`

See Netter#connect_explicit for a description of that function. Net names is an array (use square brackets to list the net names).

"connect_global" - Specifies a connection to a global net

Usage:

- `connect_global(l, name)`

See Netter#connect_global for a description of that function.

"connect_implicit" - Specifies a label pattern for implicit net connections

Usage:

- `connect_implicit(label_pattern)`

- `connect_implicit(cell_pattern, label_pattern)`

See Netter#connect_implicit for a description of that function.

"corners" - Selects corners of polygons

Usage:

- `corners([ options ]) (in condition)`
- `corners(layer [, options ])`

This function can be used with a layer argument. In this case it is equivalent to "layer.corners" (see Layer#corners). Without a layer argument, "corners" represents the corner generator/filter in primary shapes for DRC expressions (see Layer#drc and corners for more details).

Like the layer-based version, the "corners" operator accepts the output type option: "as_dots" for dot-like edges, "as_boxes" for small (2x2 DBU) box markers and "as_edge_pairs" for edge pairs. The default output type is "as_boxes".

The "corners" operator can be put into a condition which means it's applied to corners meeting a particular angle constraint.

"covering" - Selects shapes entirely covering other shapes

Usage:

- `covering(other) (optionally in condition)`

This operator represents the selector of primary shapes which entirely cover shapes from the other layer. This version can be put into a condition indicating how many shapes of the other layer need to be covered. Use this operator within DRC expressions (also see Layer#drc). If can be used as method to an expression. See there for more details: covering.

"dbu" - Gets or sets the database unit to use

Usage:

- `dbu(dbu_value)`
- `dbu`

Without any argument, this method gets the database unit used inside the DRC engine.

With an argument, sets the database unit used internally in the DRC engine. Without using that method, the database unit is automatically taken as the database unit of the last input. A specific database unit can be set in order to optimize for two layouts (i.e. take the largest common denominator). When the database unit is set, it must be set at the beginning of the script and before any operation that uses it.

"deep" - Enters deep (hierarchical) mode

Usage:

- `deep`

In deep mode, the operations will be performed in a hierarchical fashion. Sometimes this reduces the time and memory required for an operation, but this will also add some overhead for the hierarchical analysis.

"deepness" is a property of layers. Layers created with "input" while in deep mode carry hierarchy. Operations involving such layers at the only or the first argument are carried out in hierarchical mode.

Hierarchical mode has some more implications, like "merged_semantics" being implied always. Sometimes cell variants will be created.

Deep mode can be cancelled with tiles or flat.

"deep_reject_odd_polygons" - Gets or sets a value indicating whether the reject odd polygons in deep mode

Usage:

- `deep_reject_odd_polygons(flag)`
- `deep_reject_odd_polygons`

In deep mode, non-orientable (e.g. "8"-shaped) polygons may not be resolved properly. By default the interpretation of such polygons is undefined - they may even vanish entirely. By setting this flag to true, the deep mode layout processor will reject such polygons with an error.

"device_scaling" - Specifies a dimension scale factor for the geometrical device properties

Usage:

- `device_scaling(factor)`

See Netter#device_scaling for a description of that function.

"diode" - Supplies the diode extractor class

Usage:

- `diode(name)`
- `diode(name, class)`

Use this class with extract_devices to specify extraction of a planar diode

See DeviceExtractorDiode for more details about this extractor.

"dmos3" - Supplies the DMOS3 transistor extractor class

Usage:

- `dmos3(name)`
- `dmos3(name, class)`

Use this class with extract_devices to specify extraction of a three-terminal DMOS transistor. A DMOS transistor is essentially the same than a MOS transistor, but source and drain are separated.

See DeviceExtractorMOS3Transistor for more details about this extractor (strict mode applies for 'dmos3').

"dmos4" - Supplies the MOS4 transistor extractor class

Usage:

- `dmos4(name)`
- `dmos4(name, class)`

Use this class with extract_devices to specify extraction of a four-terminal DMOS transistor. A DMOS transistor is essentially the same than a MOS transistor, but source and drain are separated.

See DeviceExtractorMOS4Transistor for more details about this extractor (strict mode applies for 'dmos4').

"edge" - Creates an edge object

Usage:

- `edge(...)`

This function creates an edge object. The arguments are the same than for the DEdge constructors.

"edge_layer" - Creates an empty edge layer

Usage:

- `edge_layer`

The intention of that method is to create an empty layer which can be filled with edge objects using Layer#insert.

"edge_pairs" - Gets the edges from an original layer

Usage:

- `edge_pairs(args)`

See Source#edge_pairs for a description of that function.

"edges" - Gets the edges from an original layer

Usage:

- `edges(args)`

See Source#edges for a description of that function.

"enc" - Synonym for "enclosing"

Usage:

- `enc(...)`

"enc" is the short form for enclosing.

"enclosed" - Performs an enclosing check (other enclosing layer)

Usage:

- `enclosed(other [, options ]) (in conditions)`
- `enclosed(layer, other [, options ])`

This check verifies if the polygons of the input layer are enclosed by shapes of the other input layer by a certain distance. It has manifold options. See Layer#width for the basic options such as metrics, projection and angle constraints etc. This check also features opposite and rectangle filtering. See Layer#separation for details about opposite and rectangle error filtering.

This function is essentially the reverse of enclosing. In case of "enclosed", the other layer must be bigger than the primary layer.

Classic mode

This function can be used in classic mode with a layer argument. In this case it is equivalent to "layer.enclosed" (see Layer#enclosed).

```
# classic "enclosed" check for < 0.2 um
in = layer(1, 0)
other = layer(2, 0)
errors = enclosed(in, other, 0.2.um)
```

Universal DRC

The version without a first layer is intended for use within DRC expressions together with the "universal DRC" method Layer#drc. In this case, this function needs to be put into a condition to specify the check constraints. The other options of Layer#enclosed (e.g. metrics, projection constraints, angle limits etc.) apply to this version too:

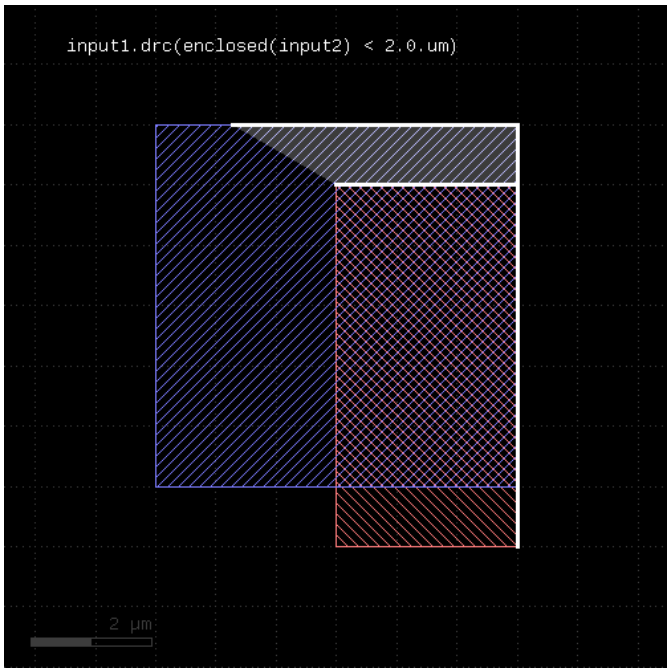
```
# universal DRC "enclosed" check for < 0.2 um
in = layer(1, 0)
other = layer(2, 0)
errors = in.drc(enclosed(other) < 0.2.um)
```

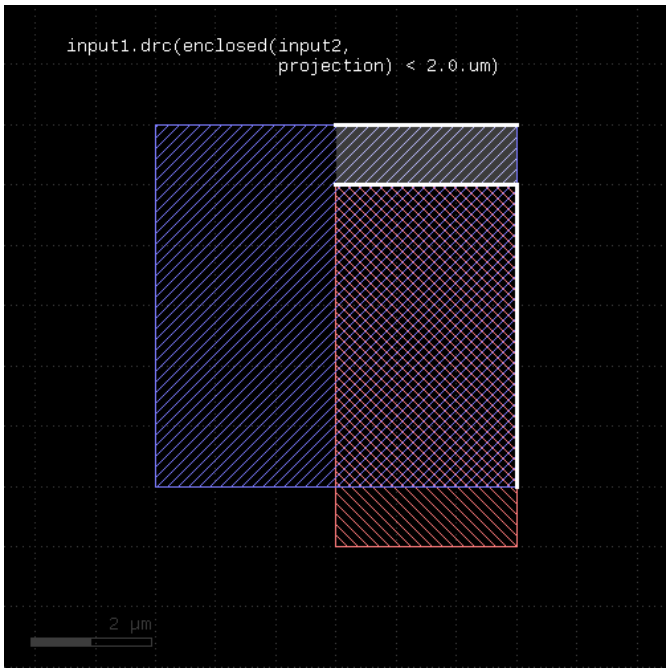
The conditions may involve an upper and lower limit. The following examples illustrate the use of this function with conditions:

```
out = in.drc(enclosed(other) < 0.2.um)
out = in.drc(enclosed(other) <= 0.2.um)
out = in.drc(enclosed(other) > 0.2.um)
out = in.drc(enclosed(other) >= 0.2.um)
out = in.drc(enclosed(other) == 0.2.um)
out = in.drc(enclosed(other) != 0.2.um)
out = in.drc(0.1.um <= enclosed(other) < 0.2.um)
```

The result of the enclosed check are edges or edge pairs forming the markers. These markers indicate the presence of the specified condition.

With a lower and upper limit, the results are edges marking the positions on the primary shape where the condition is met. With a lower limit alone, the results are edge pairs which are formed by two identical, but opposite edges attached to the primary shape. Without an upper limit only, the first edge of the marker is attached to the primary shape while the second edge is attached to the shape of the "other" layer.





When "larger than" constraints are used, this function will produce the edges from the first layer only. The result will still be edge pairs for consistency, but each edge pair holds one edge from the original polygon plus a reverse copy of that edge in the second member. Use "first_edges" to extract the actual edges from the first input (see separation for an example).

"enclosing" - Performs an enclosing check

Usage:

- `enclosing(other [, options ]) (in conditions)`
- `enclosing(layer, other [, options ])`

This check verifies if the polygons of the input layer are enclosing the shapes of the other input layer by a certain distance. It has manifold options. See Layer#width for the basic options such as metrics, projection and angle constraints etc. This check also features opposite and rectangle filtering. See Layer#separation for details about opposite and rectangle error filtering.

Classic mode

This function can be used in classic mode with a layer argument. In this case it is equivalent to "layer.enclosing" (see Layer#enclosing).

```
# classic "enclosing" check for < 0.2 um
in = layer(1, 0)
other = layer(2, 0)
errors = enclosing(in, other, 0.2.um)
```

Universal DRC

The version without a first layer is intended for use within DRC expressions together with the "universal DRC" method Layer#drc. In this case, this function needs to be put into a condition to specify the check constraints. The other options of Layer#enclosing (e.g. metrics, projection constraints, angle limits etc.) apply to this version too:

```
# universal DRC "enclosing" check for < 0.2 um
```

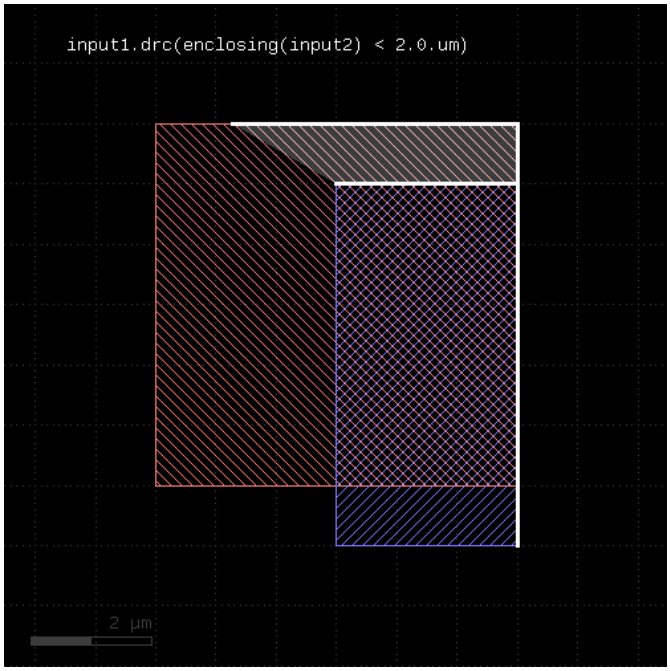
```
in = layer(1, 0)
other = layer(2, 0)
errors = in.drc(enclosing(other) < 0.2.um)
```

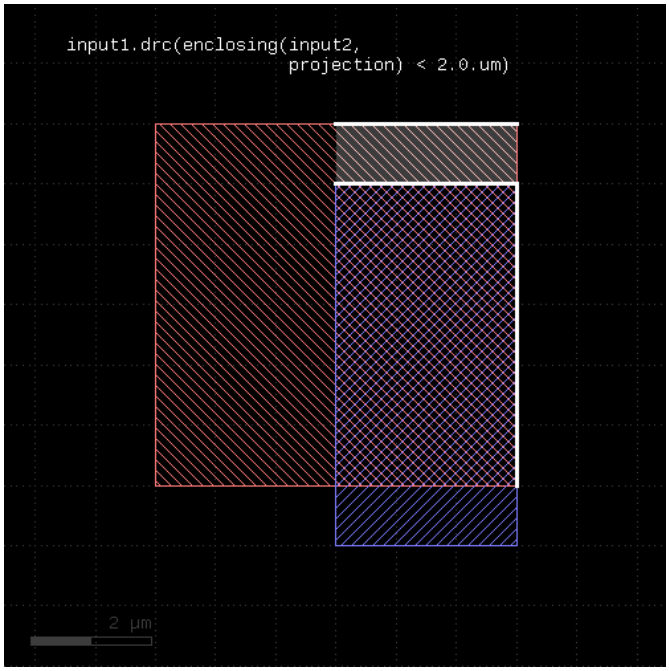
The conditions may involve an upper and lower limit. The following examples illustrate the use of this function with conditions:

```
out = in.drc(enclosing(other) < 0.2.um)
out = in.drc(enclosing(other) <= 0.2.um)
out = in.drc(enclosing(other) > 0.2.um)
out = in.drc(enclosing(other) >= 0.2.um)
out = in.drc(enclosing(other) == 0.2.um)
out = in.drc(enclosing(other) != 0.2.um)
out = in.drc(0.1.um <= enclosing(other) < 0.2.um)
```

The result of the enclosing check are edges or edge pairs forming the markers. These markers indicate the presence of the specified condition.

With a lower and upper limit, the results are edges marking the positions on the primary shape where the condition is met. With a lower limit alone, the results are edge pairs which are formed by two identical, but opposite edges attached to the primary shape. Without an upper limit only, the first edge of the marker is attached to the primary shape while the second edge is attached to the shape of the "other" layer.





When "larger than" constraints are used, this function will produce the edges from the first layer only. The result will still be edge pairs for consistency, but each edge pair holds one edge from the original polygon plus a reverse copy of that edge in the second member. Use "first_edges" to extract the actual edges from the first input (see separation for an example).

"error" - Prints an error

Usage:

- `error(message)`

Similar to log, but the message is printed formatted as an error

"extent" - Creates a new layer with the bounding box of the default source or cell bounding boxes

Usage:

- `extent`
- `extent(cell_filter)`

See Source#extent for a description of that function.

"extent_refs" - Returns partial references to the boundings boxes of the polygons

Usage:

- `extent_refs([ options ])`
- `extent_refs(layer, [ options ])`

This function can be used with a layer argument. In this case it is equivalent to "layer.extent_refs" (see Layer#extent_refs). Without a layer argument, "extent_refs" represents the partial extents extractor on primary shapes within DRC expressions (see Layer#drc and extent_refs for more details).

"extents" - Returns the bounding box of each input object

Usage:

- `extents([ enlargement ])`
- `extents(layer, [ enlargement ])`

This function can be used with a layer argument. In this case it is equivalent to "layer.extents" (see Layer#extents). Without a layer argument, "extents" represents the extents generator on primary shapes within DRC expressions (see Layer#drc and extents for more details).

"extract_devices" - Extracts devices for a given device extractor and device layer selection

Usage:

- `extract_devices(extractor, layer_hash)`
- `extract_devices(extractor_class, name, layer_hash)`

See Netter#extract_devices for a description of that function.

"flat" - Disables tiling mode

Usage:

- `flat`

Disables tiling mode. Tiling mode can be enabled again with tiles later.

"foreign" - Represents all other polygons from primary except the current one

Usage:

- `foreign`

The primary input of the universal DRC function is the layer the Layer#drc function is called on. This operation represents all "other" primary polygons while primary represents the current polygon.

This feature opens new options for processing layouts beyond the abilities of the classical DRC concept. For classic DRC, intra-layer interactions are always symmetric: a polygon cannot be considered separated from its neighbors on the same layer.

The following example computes every part of the input which is closer than 0.5 micrometers to other (disconnected) polygons on the same layer:

```
out = in.drc(primary & foreign.sized(0.5.um))
```

"global_transform" - Gets or sets a global transformation

Usage:

- `global_transform`
- `global_transform([ transformations ])`

Applies a global transformation to the default source layout. See Source#global_transform for a description of this feature.

"holes" - Selects all holes from the input polygons

Usage:

- `holes`
- `holes(layer)`

This function can be used with a layer argument. In this case it is equivalent to "layer.holes" (see Layer#hulls). Without a layer argument, "holes" represents a hole extractor for primary shapes in DRC expressions (see Layer#drc and DRC#hulls for more details).

"hulls" - Selects all hulls from the input polygons

Usage:

- `hulls`
- `hulls(layer)`

This function can be used with a layer argument. In this case it is equivalent to "layer.hulls" (see Layer#hulls). Without a layer argument, "hulls" represents a hull contour extractor for primary shapes in DRC expressions (see Layer#drc and hulls for more details).

"if_all" - Evaluates to the primary shape when all condition expression results are non-empty

Usage:

- `if_all(c1, ... cn)`

This function will evaluate the conditions c1 to cn and return the current primary shape if all conditions render a non-empty result. The following example selects all shapes which are rectangles and whose area is larger than 0.5 square micrometers:

```
out = in.drc(if_all(area > 0.5, rectangles))
```

The condition expressions may be of any type (edges, edge pairs and polygons).

"if_any" - Evaluates to the primary shape when any condition expression results is non-empty

Usage:

- `if_any(c1, ... cn)`

This function will evaluate the conditions c1 to cn and return the current primary shape if at least one condition renders a non-empty result. See if_all for an example how to use the if_... functions.

"if_none" - Evaluates to the primary shape when all of the condition expression results are empty

Usage:

- `if_none(c1, ... cn)`

This function will evaluate the conditions c1 to cn and return the current primary shape if all conditions renders an empty result. See if_all for an example how to use the if_... functions.

"ignore_extraction_errors" - Specifies whether to ignore extraction errors

Usage:

- `ignore_extraction_errors(value)`

See Netter#ignore_extraction_errors for a description of that function.

"info" - Outputs as message to the logger or progress window

Usage:

- `info(message)`
- `info(message, indent)`

Prints the message to the log window in verbose mode. In non-verbose more, nothing is printed but a statement is put into the progress window. log is a function that always prints a message.

"input" - Fetches the shapes from the specified input from the default source

Usage:

- `input(args)`

See Source#input for a description of that function. This method will fetch polygons and labels. See polygons and labels for more specific versions of this method.

"inside" - Selects shapes entirely inside other shapes

Usage:

- `inside(other)`

This operator represents the selector of primary shapes which are inside shapes from the other layer. Use this operator within DRC expressions (also see Layer#drc). If can be used as method to an expression. See there for more details: inside.

"interacting" - Selects shapes interacting with other shapes

Usage:

- `interacting(other) (optionally in condition)`

This operator represents the selector of primary shapes which interact with shapes from the other layer. This version can be put into a condition indicating how many shapes of the other layer need to be covered. Use this operator within DRC expressions (also see Layer#drc). If can be used as method to an expression. See there for more details: interacting.

"is_deep?" - Returns true, if in deep mode

Usage:

- `is_deep?`

"is_tiled?" - Returns true, if in tiled mode

Usage:

- `is_tiled?`

"iso" - Synonym for "isolated"

Usage:

- `iso(...)`

"iso" is the short form for isolated.

"isolated" - Performs an isolation (inter-polygon space) check

Usage:

- `isolated([ options ]) (in conditions)`
- `iso([ options ]) (in conditions)`
- `isolated(layer [, options ])`
- `iso(layer [, options ])`

Provides a intra-polygon space check for polygons. It is similar to space, but checks inter-polygon space only. "iso" is a synonym for "isolated". This check has manifold options. See Layer#width for the basic options such as metrics, projection and angle constraints etc. This check also features opposite and rectangle filtering. See Layer#separation for details about opposite and rectangle error filtering.

Classic mode

This function can be used in classic mode with a layer argument. In this case it is equivalent to "layer.isolated" (see Layer#isolated).

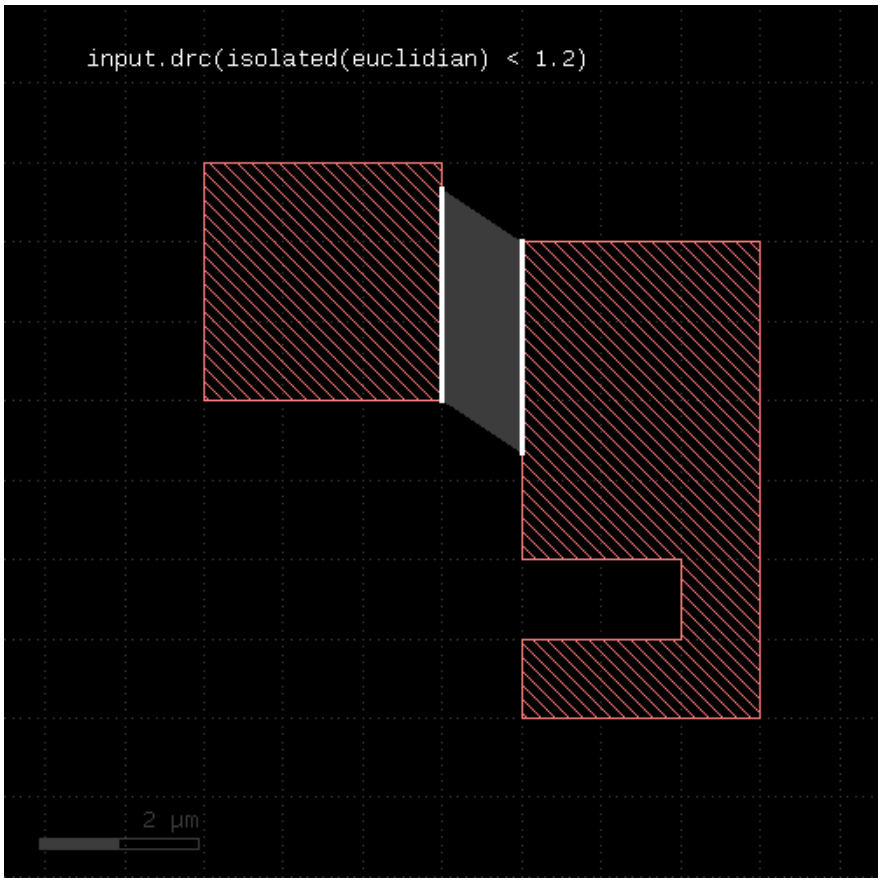
```
# classic "isolated" check for space < 1.2 um
in = layer(1, 0)
errors = isolated(in, 1.2.um)
```

Universal DRC

The version without a layer is intended for use within DRC expressions together with the "universal DRC" method Layer#drc. In this case, this function needs to be put into a condition to specify the check constraints. The other options of Layer#isolated (e.g. metrics, projection constraints, angle limits etc.) apply to this version too:

```
# universal DRC "isolated" check for space < 1.2.um
in = layer(1, 0)
errors = in.drc(isolated < 1.2.um)
```

See enclosing for more details about the various ways to specify conditions.



"l2n_data" - Gets the internal LayoutToNetlist object for the default Netter

Usage:

- `l2n_data`

See Netter#l2n_data for a description of that function.

"labels" - Gets the labels (text) from an original layer

Usage:

- `labels(args)`

See Source#labels for a description of that function.

"layers" - Gets the layers contained in the default source

Usage:

- `layers`

See Source#layers for a description of that function.

"layout" - Specifies an additional layout for the input source.

Usage:

- `layout`
- `layout(what)`

This function can be used to specify a new layout for input. It returns an Source object representing that layout. The "input" method of that object can be used to get input layers for that layout.

"what" specifies what input to use. "what" be either

- A string "@n" specifying input from a cellview in the current view
- A layout filename plus an optional cell name
- A Layout object
- A Cell object

Without any arguments the default layout is returned.

If a file name is given, a cell name can be specified as the second argument. If not, the top cell is taken which must be unique in that case.

Having specified a layout for input enables to use the input method for getting input:

```
# XOR between layers 1 or the default input and "second_layout.gds":
l2 = layout("second_layout.gds")
(input(1, 0) ^ l2.input(1, 0)).output(100, 0)
```

For further methods on the source object see Source.

"length" - Computes the total edge length of an edge layer or in universal DRC context: selects edges based on a length condition

Usage:

- `length (in condition)`
- `length(layer)`

This function can be used with a layer argument. In this case it is equivalent to "layer.length" (see Layer#length). Without a layer argument, "length" represents the edge length filter on the primary shape edges in DRC expressions (see Layer#drc and DRC#length for more details). In this context, the operation acts similar to Layer#with_length.

"log" - Outputs as message to the logger window

Usage:

- `log(message)`
- `log(message, indent)`

Prints the message to the log window. info is a function that prints a message only if verbose mode is enabled.

"log_file" - Specify the log file where to send to log to

Usage:

- `log_file(filename)`

After using that method, the log output is sent to the given file instead of the logger window or the terminal.

"make_layer" - Creates an empty polygon layer based on the hierarchical scheme selected

Usage:

- `make_layer`

The intention of this method is to provide an empty polygon layer based on the hierarchical scheme selected. This will create a new layer with the hierarchy of the current layout in deep mode and a flat layer in flat mode. This method is similar to polygon_layer, but the latter does not create a hierarchical layer. Hence the layer created by make_layer is suitable for use in device extraction for example, while the one delivered by polygon_layer is not.

On the other hand, a layer created by the make_layer method is not intended to be filled with Layer#insert.

"max_area_ratio" - Gets or sets the maximum bounding box to polygon area ratio for deep mode fragmentation

Usage:

- `max_area_ratio(ratio)`
- `max_area_ratio`

In deep mode, polygons with a bounding box to polygon area ratio bigger than the given number will be split into smaller chunks to optimize performance (which gets better if the polygon's bounding boxes do not cover a lot of empty space). The default threshold is 3.0 which means fairly compact polygons. Use this method with a numeric argument to set the value and without an argument to get the current maximum area ratio. Set the value to zero to disable splitting by area ratio.

See also max_vertex_count for the other option affecting polygon splitting.

"max_vertex_count" - Gets or sets the maximum vertex count for deep mode fragmentation

Usage:

- `max_vertex_count(count)`
- `max_vertex_count`

In deep mode, polygons with more than the given number of vertexes will be split into smaller chunks to optimize performance (which is better or less complex polygons). The default threshold is 16 vertexes. Use this method with a vertex count to set the value and without an argument to get the current maximum vertex count. Set the value to zero to disable splitting by vertex count.

See also max_area_ratio for the other option affecting polygon splitting.

"middle" - Returns the centers of polygon bounding boxes

Usage:

- `middle([ options ])`
- `middle(layer, [ options ])`

This function can be used with a layer argument. In this case it is equivalent to "layer.middle" (see Layer#middle). Without a layer argument, "middle" represents the bounding box center marker generator on primary shapes within DRC expressions (see Layer#drc and middle for more details).

"mos3" - Supplies the MOS3 transistor extractor class

Usage:

- `mos3(name)`
- `mos3(name, class)`

Use this class with extract_devices to specify extraction of a three-terminal MOS transistor.

See DeviceExtractorMOS3Transistor for more details about this extractor (non-strict mode applies for 'mos3').

"mos4" - Supplies the MOS4 transistor extractor class

Usage:

- `mos4(name)`
- `mos4(name, class)`

Use this class with extract_devices to specify extraction of a four-terminal MOS transistor.

See DeviceExtractorMOS4Transistor for more details about this extractor (non-strict mode applies for 'mos4').

"netlist" - Obtains the extracted netlist from the default Netter

The netlist is a Netlist object. If no netlist is extracted yet, this method will trigger the extraction process. See Netter#netlist for a description of this function.

"netter" - Creates a new netter object

Usage:

- `netter`

See Netter for more details

"new_report" - Creates a new report database object for use in "output"

Usage:

- `new_report(description [, filename [, cellname ] ])`

This method creates an independent report object. This object can be used in "output" to send a layer to a different report than the default report or target.

Arguments are the same than for report.

See Layer#output for details about this feature.

"new_target" - Creates a new layout target object for use in "output"

Usage:

- `new_target(what [, cellname])`

This method creates an independent target object. This object can be used in "output" to send a layer to a different layout file than the default report or target.

Arguments are the same than for target.

See Layer#output for details about this feature.

"no_borders" - Reset the tile borders

Usage:

- `no_borders`

Resets the tile borders - see tile_borders for a description of tile borders.

"notch" - Performs a notch (intra-polygon space) check

Usage:

- `notch([ options ]) (in conditions)`
- `notch(layer [, options ])`

Provides a intra-polygon space check for polygons. It is similar to space, but checks intra-polygon space only. It has manifold options. See Layer#width for the basic options such as metrics, projection and angle constraints etc.

Classic mode

This function can be used in classic mode with a layer argument. In this case it is equivalent to "layer.notch" (see Layer#notch).

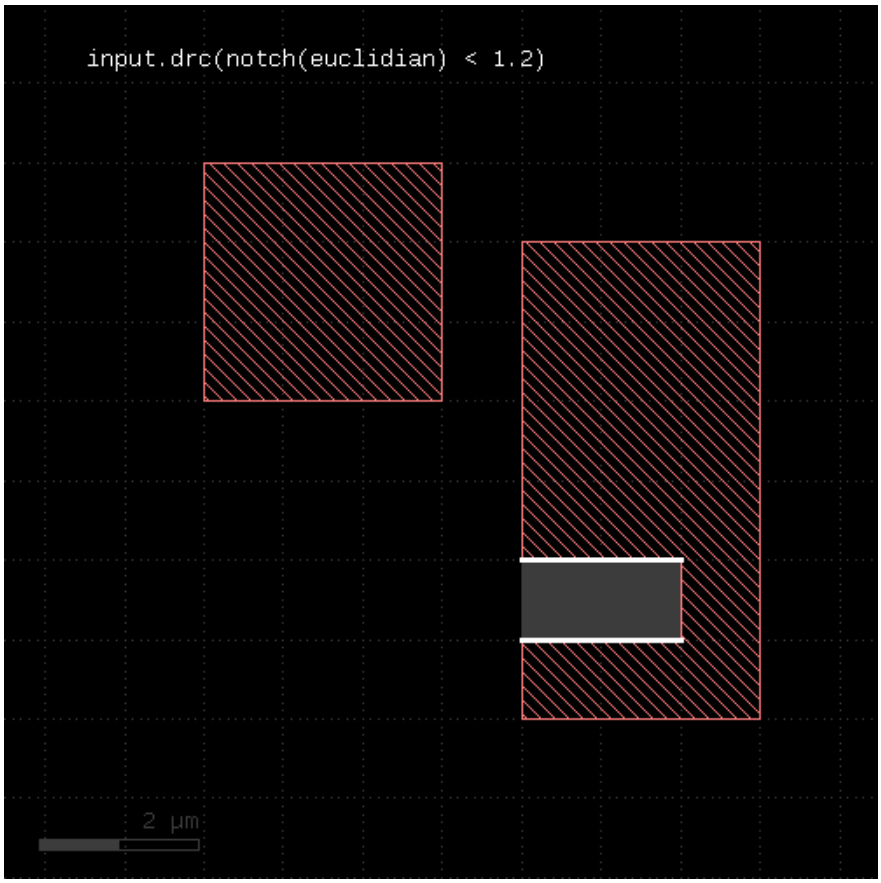
```
# classic "notch" check for space < 1.2 um
in = layer(1, 0)
errors = notch(in, 1.2.um)
```

Universal DRC

The version without a layer is intended for use within DRC expressions together with the "universal DRC" method Layer#drc. In this case, this function needs to be put into a condition to specify the check constraints. The other options of Layer#notch (e.g. metrics, projection constraints, angle limits etc.) apply to this version too:

```
# universal DRC "notch" check for space < 1.2.um
in = layer(1, 0)
errors = in.drc(notch < 1.2.um)
```

See enclosing for more details about the various ways to specify conditions.



"output" - Outputs a layer to the report database or output layout

Usage:

- `output(layer, args)`

This function is equivalent to "layer.output(args)". See Layer#output for details about this function.

"output_cell" - Specifies a target cell, but does not change the target layout

Usage:

- `output_cell(cellname)`

This method switches output to the specified cell, but does not change the target layout nor does it switch the output channel to layout if is report database.

"outside" - Selects shapes entirely outside other shapes

Usage:

- `outside(other)`

This operator represents the selector of primary shapes which are outside shapes from the other layer. Use this operator within DRC expressions (also see Layer#drc). If can be used as method to an expression. See there for more details: outside.

"overlap" - Performs an overlap check

Usage:

- `overlap(other [, options ]) (in conditions)`
- `overlap(layer, other [, options ])`

Provides an overlap check (primary layer vs. another layer). This check has manifold options. See Layer#width for the basic options such as metrics, projection and angle constraints etc. This check also features opposite and rectangle filtering. See Layer#separation for details about opposite and rectangle error filtering.

Classic mode

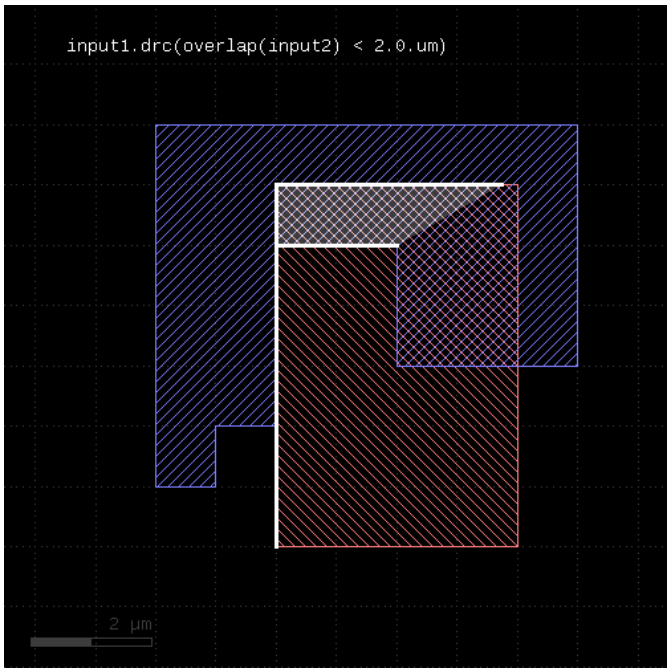
Like other checks, this function is available as a classic DRC function with a layer as the first argument and as an DRC expression operator for use with Layer#drc.

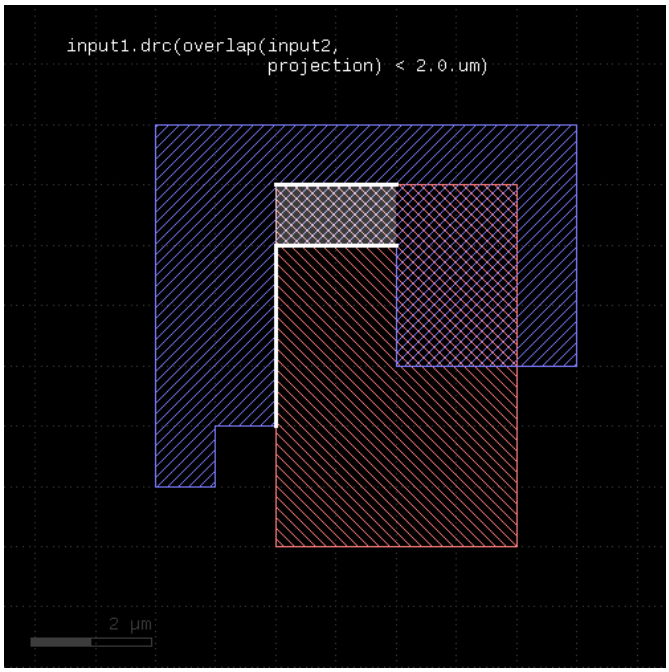
```
# classic "overlap" check for < 0.2 um
in = layer(1, 0)
other = layer(2, 0)
errors = overlap(in, other, 0.2.um)
```

Universal DRC

For use with the "unversal DRC" put the separation expression into the "drc" function call and use a condition to specify the constraint:

```
# universal DRC "overlap" check for < 0.2 um
in = layer(1, 0)
other = layer(2, 0)
errors = in.drc(overlap(other) < 0.2.um)
```





When "larger than" constraints are used, this function will produce the edges from the first layer only. The result will still be edge pairs for consistency, but each edge pair holds one edge from the original polygon plus a reverse copy of that edge in the second member. Use "first_edges" to extract the actual edges from the first input (see separation for an example).

"overlapping" - Selects shapes overlapping with other shapes

Usage:

- `overlapping(other) (optionally in condition)`

This operator represents the selector of primary shapes which overlap shapes from the other layer. This version can be put into a condition indicating how many shapes of the other layer need to be covered. Use this operator within DRC expressions (also see Layer#drc). If can be used as method to an expression. See there for more details: overlapping.

"p" - Creates a point object

Usage:

- `p(x, y)`

A point is not a valid object by itself, but it is useful for creating paths for polygons:

```
x = polygon_layer
x.insert(polygon([ p(0, 0), p(16.0, 0), p(8.0, 8.0) ]))
```

"path" - Creates a path object

Usage:

- `path(...)`

This function creates a path object. The arguments are the same than for the DPath constructors.

"perimeter" - Computes the total perimeter or in universal DRC context: selects the primary shape if the perimeter is meeting the condition

Usage:

- `perimeter (in condition)`
- `perimeter(layer)`

This function can be used with a layer argument. In this case it is equivalent to "layer.perimeter" (see Layer#perimeter) and returns the total perimeter of all polygons in the layer.

Without a layer argument, "perimeter" represents a perimeter filter for primary shapes in DRC expressions (see Layer#drc and DRC#perimeter for more details).

"polygon" - Creates a polygon object

Usage:

- `polygon(...)`

This function creates a polygon object. The arguments are the same than for the DPolygon constructors.

"polygon_layer" - Creates an empty polygon layer

Usage:

- `polygon_layer`

The intention of that method is to create an empty layer which can be filled with polygon-like objects using Layer#insert. A similar method which creates a hierarchical layer in deep mode is make_layer. This other layer is better suited for use with device extraction.

"polygons" - Fetches the polygons (or shapes that can be converted to polygons) from the specified input from the default source

Usage:

- `polygons(args)`

See Source#polygons for a description of that function.

"primary" - Represents the primary input of the universal DRC function

Usage:

- `primary`

The primary input of the universal DRC function is the layer the Layer#drc function is called on.

"profile" - Profiles the script and provides a runtime + memory statistics

Usage:

- `profile`
- `profile(n)`

Turns profiling on or off (default). In profiling mode, the system will collect statistics about rules executed, their execution time and memory information. The argument specifies how many operations to print at the end of the run. Without an argument, all operations are printed. Passing "false" for the argument will disable profiling. This is the default.

"props_copy" - Specifies "copy properties" on operations supporting user properties constraints

This properties constraint does not constrain the operation, but instructs it to attach the properties from the primary input to the output objects.

See also props_ne and props_eq.

"props_eq" - Specifies "same properties" for operations supporting user properties constraints

Some operations such as boolean AND support properties constraints. By giving a "props_eq" constraint, the operation is performed only on shapes with the same properties, where "properties" stands for the full set of key/value pairs.

Note that you have to enable properties explicitly or generate properties (e.g. with the DRCLayer#nets method).

Example:

```
connect(metal1, via1)
connect(via1, metal2)
... further connect statements

m1m2_overlap_connected = metal1.nets.and(metal2, props_eq)
```

"props_eq" can be combined with props_copy. In this case, properties are transferred to the output shapes and can be used in further processing:

```
m1m2_overlap_connected = metal1.nets.and(metal2, props_eq + props_copy)
```

See also props_ne.

"props_ne" - Specifies "different properties" for operations supporting user properties constraints

Some operations such as boolean AND support properties constraints. By giving a "props_ne" constraint, the operation is performed only on shapes with different properties, where "properties" stands for the full set of key/value pairs.

Note that you have to enable properties explicitly or generate properties (e.g. with the DRCLayer#nets method).

Example:

```
connect(metal1, via1)
connect(via1, metal2)
... further connect statements

m1m2_overlap_not_connected = metal1.nets.and(metal2, props_ne)
```

"props_ne" can be combined with props_copy. In this case, properties are transferred to the output shapes and can be used in further processing:

```
m1m2_overlap_connected = metal1.nets.and(metal2, props_ne + props_copy)
```

See also props_eq.

"rectangles" - Selects all polygons which are rectangles

Usage:

- `rectangles`
- `rectangles(layer)`

This function can be used with a layer argument. In this case it is equivalent to "layer.rectangles" (see Layer#rectangles). Without a layer argument, "rectangles" represents the rectangles filter for primary shapes in DRC expressions (see Layer#drc and rectangles for more details).

"rectilinear" - Selects all polygons which are rectilinear

Usage:

- `rectilinear`
- `rectilinear(layer)`

This function can be used with a layer argument. In this case it is equivalent to "layer.rectilinear" (see Layer#rectilinear). Without a layer argument, "rectilinear" represents the rectilinear polygons filter for primary shapes in DRC expressions (see Layer#drc and rectilinear for more details).

"region_overlap" - Specifies region selected input in "overlap mode"

Usage:

- `region_overlap(args)`

See Source#overlapping for a description of that function.

The following code will select shapes overlapping a 500x600 micron rectangle (lower left corner at 0,0) from the input layout. The shapes will not be clipped:

```
region_overlapping(0.mm, 0.mm, 0.5.mm, 0.6.mm)
# shapes will now be the ones overlapping the rectangular region
l1 = input(1, 0)
```

To remove this condition, call "region_overlapping" without any arguments.

"region_touch" - Specifies region selected input in "touch mode"

Usage:

- `region_touch(args)`

See Source#touching for a description of that function.

The following code will select shapes touching a 500x600 micron rectangle (lower left corner at 0,0) from the input layout. The shapes will not be clipped:

```
region_touch(0.mm, 0.mm, 0.5.mm, 0.6.mm)
# shapes will now be the ones touching the rectangular region
l1 = input(1, 0)
```

To remove this condition, call "region_touch" without any arguments.

"relative_height" - Selects primary shapes based on the ratio of height and width of their bounding boxes

Usage:

- `relative_height (in condition)`

See Layer#drc, relative_height and DRC#relative_height for more details.

"report" - Specifies a report database for output

Usage:

- `report(description [, filename [, cellname ] ])`

After specifying a report database for output, output method calls are redirected to the report database. The format of the output calls changes and a category name plus description can be specified rather than a layer/datatype number of layer name. See the description of the output method for details.

If a filename is given, the report database will be written to the specified file name. Otherwise it will be shown but not written.

If external input is specified with source, "report" must be called after "source".

The cellname specifies the top cell used for the report file. By default this is the cell name of the default source. If there is no source layout you'll need to give the cell name in the third parameter.

"report_netlist" - Specifies an extracted netlist report for output

Usage:

- `report_netlist([ filename [, long ] ])`

This method applies to runsets creating a netlist through extraction. Extraction happens when connections and/or device extractions are made. If this statement is used, the extracted netlist plus the net and device shapes are turned into a layout-to-netlist report (L2N database) and shown in the netlist browser window. If a file name is given, the report will also be written to the given file. If a file name is given and "long" is true, a verbose version of the L2N DB format will be used.

"resistor" - Supplies the resistor extractor class

Usage:

- `resistor(name, sheet_rho)`
- `resistor(name, sheet_rho, class)`

Use this class with extract_devices to specify extraction of a resistor.

The sheet_rho value is the sheet resistance in ohms/square. It is used to compute the resistance from the geometry.

See DeviceExtractorResistor for more details about this extractor.

"resistor_with_bulk" - Supplies the resistor extractor class that includes a bulk terminal

Usage:

- `resistor_with_bulk(name, sheet_rho)`
- `resistor_with_bulk(name, sheet_rho, class)`

Use this class with extract_devices to specify extraction of a resistor with a bulk terminal. The sheet_rho value is the sheet resistance in ohms/square.

See DeviceExtractorResistorWithBulk for more details about this extractor.

"rounded_corners" - Applies corner rounding

Usage:

- `rounded_corners(inner, outer, n)`
- `rounded_corners(layer, inner, outer, n)`

This function can be used with a layer argument. In this case it is equivalent to "layer.rounded_corners" (see Layer#rounded_corners). Without a layer argument, "rounded_corners" represents the corner rounding algorithm on primary shapes within DRC expressions (see Layer#drc and rounded_corners for more details).

"secondary" - Provides secondary input for the "drc" universal DRC function

Usage:

- `secondary(layer)`

To supply additional input for the universal DRC expressions (see Layer#drc), use "secondary" with a layer argument. This example provides a boolean AND between l1 and l2:

```
l1 = layer(1, 0)
l2 = layer(2, 0)
out = l1.drc(primary & secondary(l2))
```

"select" - Specifies cell filters on the default source

Usage:

- `select(args)`

See Source#select for a description of that function. Using the global version does not create a new source, but modifies the default source.

```
# Selects only B cell instances below the top cell
select("-", "+B*")
l1 = input(1, 0)
```

"sep" - Synonym for "separation"

Usage:

- `sep(...)`

"sep" is the short form for separation.

"separation" - Performs a separation check

Usage:

- `separation(other [, options ]) (in conditions)`

- `separation(layer, other [, options ])`

Provides a separation check (primary layer vs. another layer). Like enclosing this function provides a two-layer check, but checking the distance rather than the overlap. This check has manifold options. See Layer#width for the basic options such as metrics, projection and angle constraints etc. This check also features opposite and rectangle filtering. See Layer#separation for details about opposite and rectangle error filtering.

Classic mode

Like enclosing, this function is available as a classic DRC function with a layer as the first argument and as an DRC expression operator for use with Layer#drc.

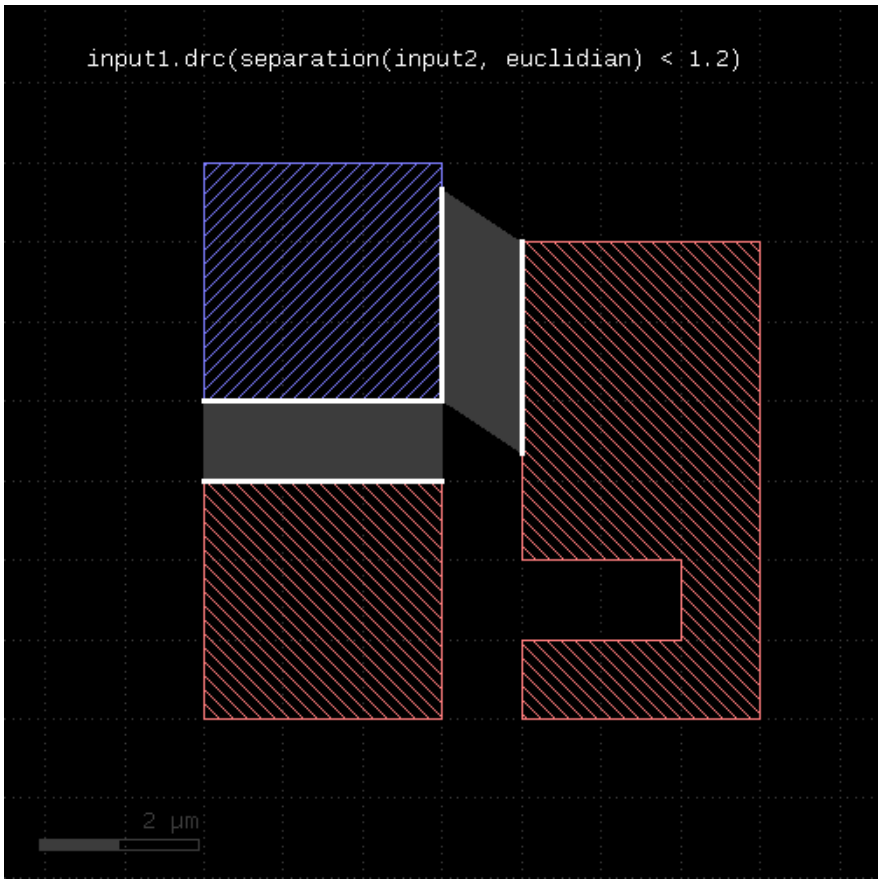
```
# classic "separation" check for distance < 0.2 um
in = layer(1, 0)
other = layer(2, 0)
errors = separation(in, other, 0.2.um)
```

Universal DRC

For use with the "universal DRC" put the separation expression into the "drc" function call and use a condition to specify the constraint:

```
# universal DRC "separation" check for distance < 0.2 um
in = layer(1, 0)
other = layer(2, 0)
errors = in.drc(separation(other) < 0.2.um)
```

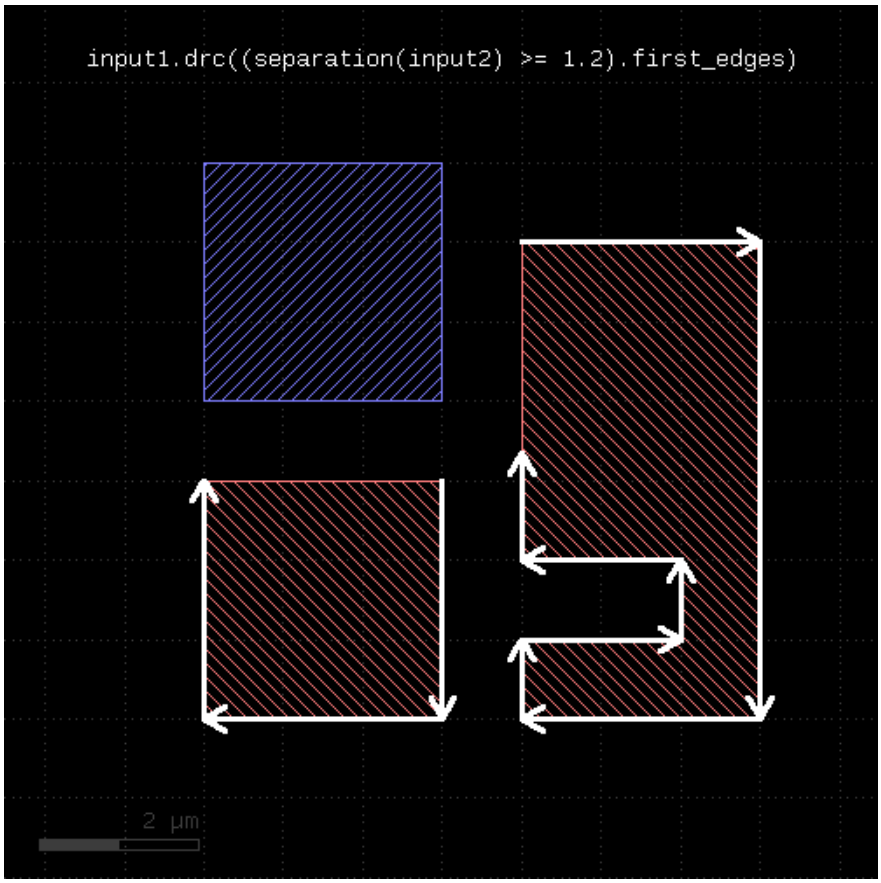
enclosing explains the constraints and how the work in generating error markers.



When "larger than" constraints are used, this function will produce the edges from the first layer only. The result will still be edge pairs for consistency, but each edge pair holds one edge from the original polygon plus a reverse copy of that edge in the second member. Use "first_edges" to extract the actual edges from the first input:

```
l1_edges_without_l2 = l1.drc((separation(l2) >= 1.0).first_edges)
```

The following image shows the effect of such a negative-output separation check:



"silent" - Resets verbose mode

Usage:

- `silent`

This function is equivalent to "verbose(false)" (see verbose)

"sized" - Returns the sized version of the input

Usage:

- `sized(d [, mode])`
- `sized(dx, dy [, mode]))`
- `sized(layer, d [, mode])`
- `sized(layer, dx, dy [, mode]))`

This function can be used with a layer argument. In this case it is equivalent to "layer.sized" (see Layer#sized). Without a layer argument, "sized" represents the polygon sizer on primary shapes within DRC expressions (see Layer#drc and sized for more details).

"smoothed" - Applies smoothing

Usage:

- `smoothed(d)`
- `smoothed(layer, d)`

This function can be used with a layer argument. In this case it is equivalent to "layer.smoothed" (see Layer#smoothed). Without a layer argument, "smoothed" represents the polygon smoother on primary shapes within DRC expressions (see Layer#drc and smoothed for more details).

"soft_connect" - Specifies a soft connection between two layers

Usage:

- `soft_connect(a, b)`

A "soft connection" is made between two layers and is a directional connection (like an ideal diode). Soft connections allow detecting if nets are connected via a high-ohmic substrate or diffusion layer (the "lower" layer). "b" is the "lower" and "a" the upper layer.

See Netter#connect for a more detailed description of that function.

"soft_connect_global" - Specifies a soft connection to a global net

Usage:

- `soft_connect_global(l, name)`

Like soft_connect, a soft connection is made between a layer and a global net (e.g. substrate). The global net is always the "lower" net of the soft connection.

See Netter#soft_connect_global for a more detailed description of that function.

"source" - Specifies a source layout

Usage:

- `source`
- `source(what)`

This function replaces the default source layout by the specified file. If this function is not used, the currently active layout is used as input.

layout is a similar method which specifies *an additional* input layout.

"what" specifies what input to use. "what" be either

- A string "@n" specifying input from a layout in the current panel
- A layout filename plus an optional cell name
- A Layout object plus an optional cell name
- A Cell object

Without any arguments the default layout is returned. If a filename is given, a cell name can be specified as the second argument. If none is specified, the top cell is taken which must be unique in that case.

```
# XOR between layers 1 of "first_layout.gds" and "second_layout.gds" and sends the results to
 "xor_layout.gds":
```

```
target("xor_layout.gds")
source("first_layout.gds")
l2 = layout("second_layout.gds")
(input(1, 0) ^ l2.input(1, 0)).output(100, 0)
```

For further methods on the source object see Source.

"space" - Performs a space check

Usage:

- `space([ options ]) (in conditions)`
- `space(layer [, options ])`

"space" looks for spacing violations between edges of the same polygon (intra-polygon checks) and between different polygons (inter-polygon checks). notch is similar function that provides only intra-polygon space checks. isolated is the version checking inter-polygon distance only. The check has manifold options. See Layer#width for the basic options such as metrics, projection and angle constraints etc.

Classic mode

This function can be used in classic mode with a layer argument. In this case it is equivalent to "layer.space" (see Layer#space). In this mode, "space" is applicable to edge layers too.

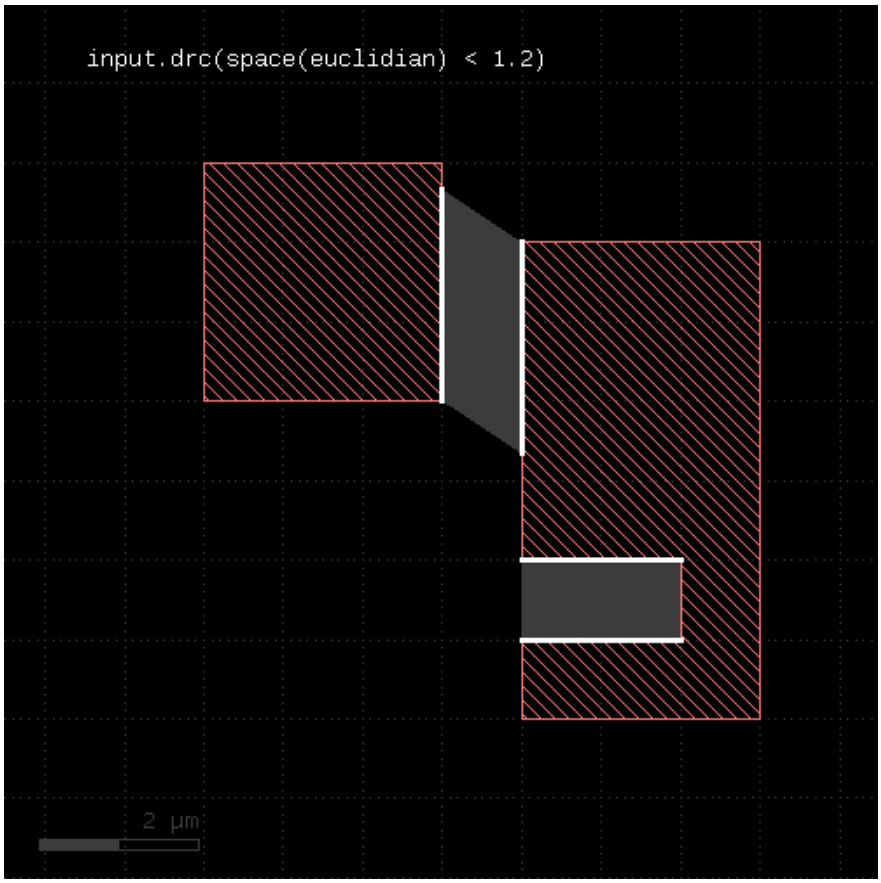
```
# classic "space" check for space < 0.2 um
in = layer(1, 0)
errors = space(in, 0.2.um)
```

Universal DRC

The version without a layer is intended for use within DRC expressions together with the "universal DRC" method Layer#drc. In this case, this function needs to be put into a condition to specify the check constraints. The other options of Layer#space (e.g. metrics, projection constraints, angle limits etc.) apply to this version too:

```
# universal DRC check for space < 0.2.um
in = layer(1, 0)
errors = in.drc(space < 0.2.um)
```

See enclosing for more details about the various ways to specify conditions.



"squares" - Selects all polygons which are squares

Usage:

- `squares`
- `squares(layer)`

This function can be used with a layer argument. In this case it is equivalent to "layer.squares" (see Layer#squares). Without a layer argument, "squares" represents the rectangles filter for primary shapes in DRC expressions (see Layer#drc and squares for more details).

"switch" - A conditional selector for the "drc" universal DRC function

Usage:

- `switch(...)`

This function provides a conditional selector for the "drc" function. It is used this way:

```
out = in.drc(switch(c1, r1, c2, r2, ..., cn, rn)
out = in.drc(switch(c1, r1, c2, r2, ..., cn, rn, rdef)
```

This function will evaluate c1 which is a universal DRC expression (see Layer#drc). If the result is not empty, "switch" will evaluate and return r1. Otherwise it will continue with c2 and the result of this expression is not empty it will return r2. Otherwise it will continue with c3/r3 etc.

If an odd number of arguments is given, the last expression is evaluated if none of the conditions c1..cn gives a non-empty result.

As a requirement, the result types of all r1..rn expressions and the rdef needs to be the same - i.e. all need to render polygons or edges or edge pairs.

"target" - Specify the target layout

Usage:

- `target(what [, cellname])`

This function can be used to specify a target layout for output. Subsequent calls of "output" will send their results to that target layout. Using "target" will disable output to a report database. If any target was specified before, that target will be closed and a new target will be set up.

"what" specifies what input to use. "what" be either

- A string "@n" (n is an integer) specifying output to a layout in the current panel
- A string "@+" specifying output to a new layout in the current panel
- A layout filename
- A Layout object
- A Cell object

Except if the argument is a Cell object, a cellname can be specified stating the cell name under which the results are saved. If no cellname is specified, either the current cell or "TOP" is used.

"target_netlist" - With this statement, an extracted netlist is finally written to a file

Usage:

- `target_netlist(filename [, format [, comment ] ])`

This method applies to runsets creating a netlist through extraction. Extraction happens when connections and/or device extractions are made. If this statement is used, the extracted netlist is written to the given file.

The format parameter specifies the writer to use. You can use nil to use the standard format or produce a SPICE writer with write_spice. See write_spice for more details.

"threads" - Specifies the number of CPU cores to use in tiling mode

Usage:

- `threads(n)`
- `threads`

If using threads, tiles are distributed on multiple CPU cores for parallelization. Still, all tiles must be processed before the operation proceeds with the next statement.

Without an argument, "threads" will return the current number of threads

"tile_borders" - Specifies a minimum tile border

Usage:

- `tile_borders(b)`
- `tile_borders(bx, by)`

The tile border specifies the distance to which shapes are collected into the tile. In order words, when processing a tile, shapes within the border distance participate in the operations.

For some operations such as booleans (Layer#and, Layer#or, ...), Layer#size and the DRC functions (Layer#width, Layer#space, ...) a tile border is automatically established. For other operations such as Layer#with_area or Layer#edges, the exact distance is unknown, because such operations may have a long range. In that cases, no border is used. The tile_borders function may be used to specify a minimum border which is used in that case. That allows taking into account at least shapes within the given range, although not necessarily all.

To reset the tile borders, use no_borders or "tile_borders(nil)".

"tiles" - Specifies tiling

Usage:

- `tiles(t)`
- `tiles(w, h)`

Specifies tiling mode. In tiling mode, the DRC operations are evaluated in tiles with width w and height h. With one argument, square tiles with width and height t are used.

Special care must be taken when using tiling mode, since some operations may not behave as expected at the borders of the tile. Tiles can be made overlapping by specifying a tile border dimension with tile_borders. Some operations like sizing, the DRC functions specify a tile border implicitly. Other operations without a defined range won't do so and the consequences of tiling mode can be difficult to predict.

In tiling mode, the memory requirements are usually smaller (depending on the choice of the tile size) and multi-CPU support is enabled (see threads). To disable tiling mode use flat or deep.

Tiling mode will disable deep mode (see deep).

"top_level" - Specifies that the circuit is a chip top level circuit

Usage:

- `top_level(flag)`

See Netter#top_level for a description of that function.

"verbose" - Sets or resets verbose mode

Usage:

- `verbose`
- `verbose(m)`

In verbose mode, more output is generated in the log file

"verbose?" - Returns true, if verbose mode is enabled

Usage:

- `verbose?`

In verbose mode, more output is generated in the log file

"warn" - Prints a warning

Usage:

- `warn(message)`

Similar to log, but the message is printed formatted as a warning

"width" - Performs a width check

Usage:

- `width([ options ]) (in conditions)`
- `width(layer [, options ])`

A width check is a check for the distance of edges of the same polygon.

Classic mode

This function can be used in classic mode with a layer argument. In this case it is equivalent to "layer.width" (see Layer#width).

```
# classic "width" check for width < 2 um
in = layer(1, 0)
errors = width(in, 0.2.um)
```

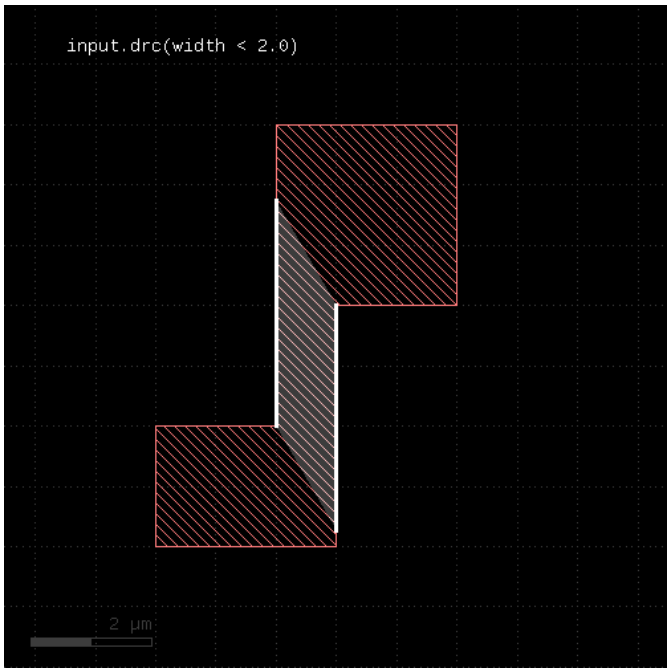
Universal DRC

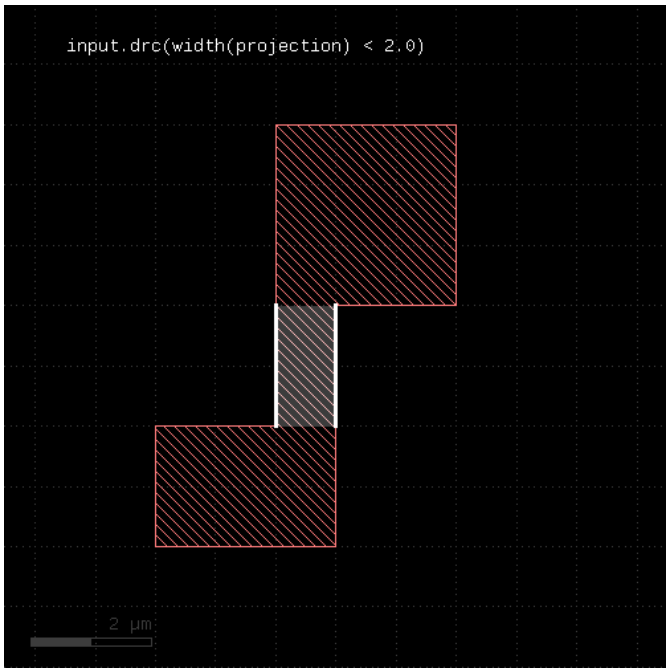
The version without a layer is intended for use within DRC expressions together with the "universal DRC" method Layer#drc. In this case, this function needs to be put into a condition to specify the check constraints. The other options of Layer#width (e.g. metrics, projection constraints, angle limits etc.) apply to this version too:

```
# universal DRC check for width < 2 um
in = layer(1, 0)
errors = in.drc(width < 0.2.um)
```

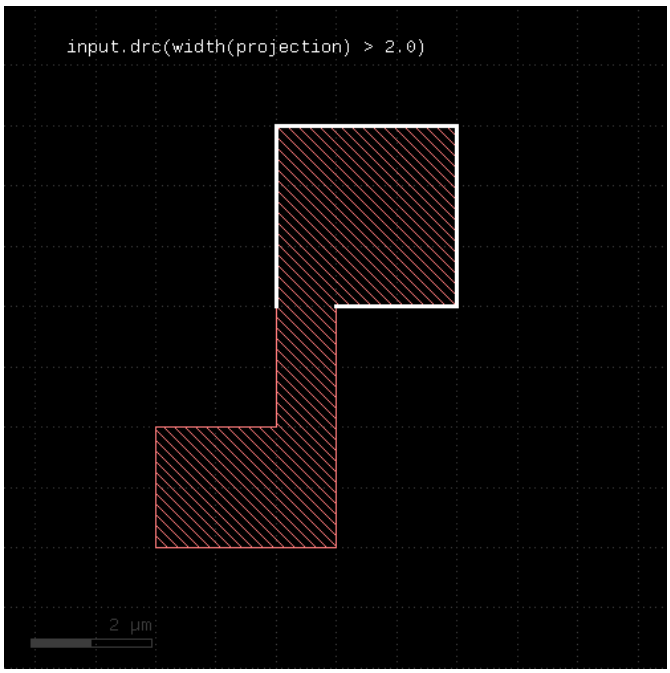
The conditions may involve an upper and lower limit. The following examples illustrate the use of this function with conditions:

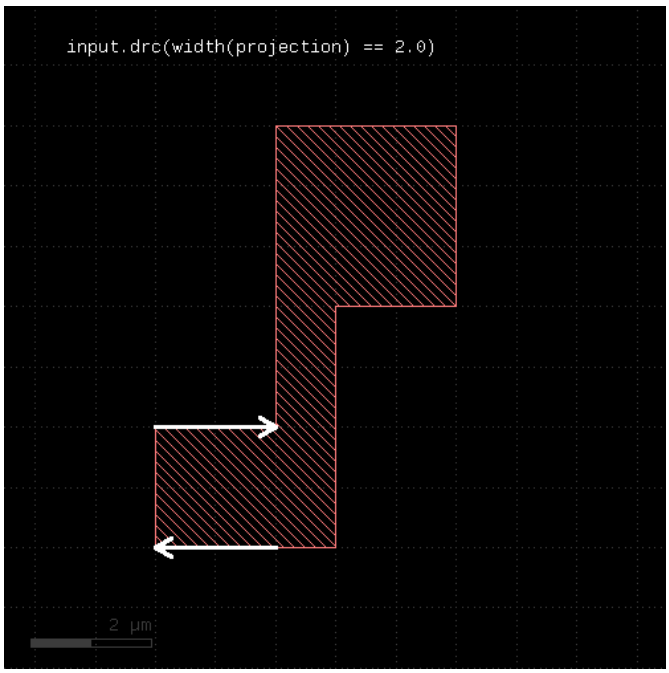
```
errors = in.drc(width < 0.2.um)
errors = in.drc(width <= 0.2.um)
errors = in.drc(width > 0.2.um)
errors = in.drc(width >= 0.2.um)
errors = in.drc(width == 0.2.um)
errors = in.drc(width != 0.2.um)
errors = in.drc(0.1.um <= width < 0.2.um)
```

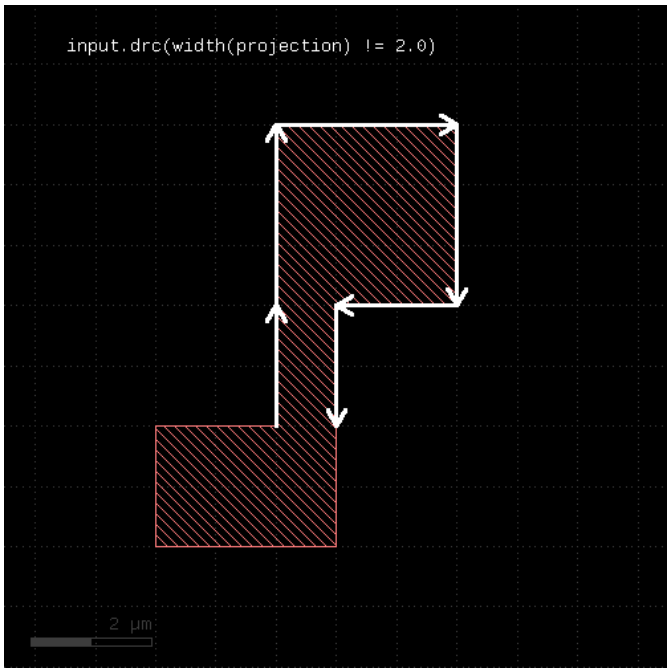


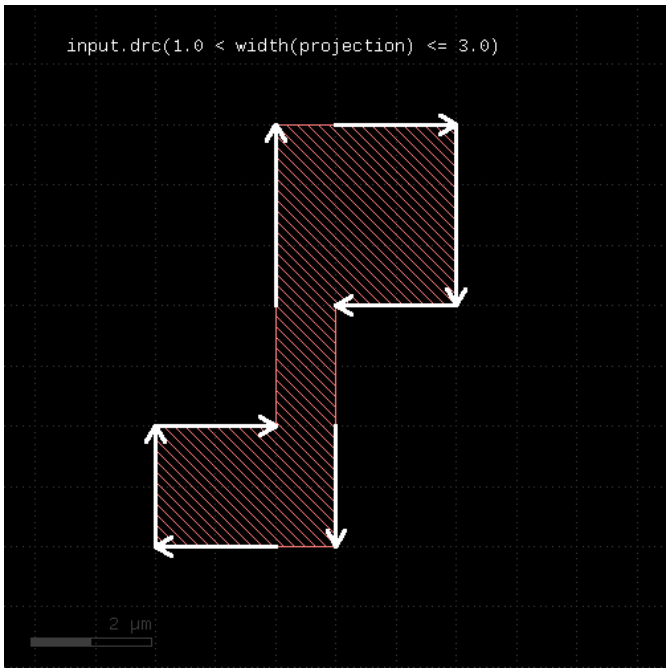


With a lower and upper limit or with the "equal" condition, the results are edges marking the positions on the primary shape where the condition is met. With a lower limit alone, the results are edge pairs which are formed by two identical, but opposite edges attached to the primary shape. Without an upper limit only, both edges are attached to different sides of the primary shape.









"with_holes" - Selects all input polygons according to their number of holes in DRC expressions

Usage:

- `with_holes (in condition)`

"with_holes" represents a polygon selector for DRC expressions selecting polygons of the primary by their number of holes (see Layer#drc and with_holes for more details).

"write_spice" - Defines SPICE output format (with options)

Usage:

- `write_spice([ use_net_names [, with_comments ] ])`
- `write_spice(writer_delegate [, use_net_names [, with_comments ] ])`

Use this option in target_netlist for the format parameter to specify SPICE format. "use_net_names" and "with_comments" are boolean parameters indicating whether to use named nets (numbers if false) and whether to add information comments such as instance coordinates or pin names.

"writer_delegate" allows using a NetlistSpiceWriterDelegate object to control the actual writing.

2.21. LVS Reference

2.21.1. LVS Reference: Netter object

The Netter object provides services related to network extraction from a layout plus comparison against a reference netlist. Similar to the DRC DRC::Netter (which lacks the compare ability), the relevant method of this object are available as global functions too where they act on a default incarnation. Usually it's not required to instantiate a Netter object explicitly.

The LVS Netter object inherits all methods of the DRC::Netter.

An individual netter object can be created, if the netter results need to be kept for multiple extractions. If you really need a Netter object, use the global netter function:

```
# create a new Netter object:
nx = netter

# build connectivity
nx.connect(poly, contact)
...

# read the reference netlist
nx.schematic("reference.cir")

# configure the netlist compare
nx.same_circuits("A", "B")
...

# runs the compare
if ! nx.compare
puts("no equivalence!")
end
```

"align" - Aligns the extracted netlist vs. the schematic

Usage:

- `align`

The align method will modify the netlists in case of missing corresponding circuits. It will flatten these circuits, thus improving the equivalence between the netlists. Top level circuits are not flattened.

This feature is in particular useful to remove structural cells like device PCells, reuse blocks etc.

This method will also remove schematic circuits for which there is no corresponding layout cell. In the extreme case of flat layout this will result in a flat vs. flat compare.

"netlist.flatten_circuit(...)" or "schematic.flatten_circuit(...)" are other (explicit) ways to flatten circuits.

Please note that flattening circuits has some side effects such as loss of details in the cross reference and net layout.

"blank_circuit" - Removes the content from the given circuits (blackboxing)

Usage:

- `blank_circuit(circuit_filter)`

This method will erase all content from the circuits matching the filter. The filter is a glob expression.

This has the following effects:

- The circuits are no longer compared (netlist vs. schematic)
- Named pins are required to match (use labels on the nets to name pins in the layout)

- Unnamed pins are treated as equivalent and can be swapped
- The selected circuits will not be purged on netlist simplification

Using this method can be useful to reduce the verification overhead for blocks which are already verifified by other ways or for which no schematic is available - e.g. hard macros.

Example:

```
# skips all MEMORY* circuits from compare
blank_circuit("MEMORY*")
```

"compare" - Compares the extracted netlist vs. the schematic

Usage:

- `compare`

Before using this method, a schematic netlist has to be loaded with schematic. The compare can be configured in more details using same_nets, same_circuits, same_device_classes and equivalent_pins.

The compare method will also modify the netlists in case of missing corresponding circuits: the unpaired circuit will be flattened then.

This method will return true, if the netlists are equivalent and false otherwise.

"consider_net_names" - Indicates whether the netlist comparer shall use net names

Usage:

- `consider_net_names(f)`

If this value is set to true (the default), the netlist comparer will employ net names to resolve ambiguities. If set to false, ambiguities will be resolved based on the topology alone. Topology resolution is more expensive.

"disable_parameter" - Indicates whether to disable a specific parameter for a given device

Usage:

- `disable_parameter(device_class_name, parameter_name)`

Disabling a parameter is the inverse of enable_parameter. Disabling a parameter will reset the "primary" flag of the parameter. This has several effects - e.g. the parameter will not be used in device compare during netlist matching by default.

This is not a strong concept but rather a hint for the system. Disabling a parameter for netlist compare without side effects is possible with the ignore_parameter function. In the same way, tolerance will enable a parameter for netlist compare regardless of the "primary" status of the parameter.

"enable_parameter" - Indicates whether to enable a specific parameter for a given device

Usage:

- `enable_parameter(device_class_name, parameter_name)`

The parameter is made "primary" which enables further applications - e.g. it is netlisted for some elements which normally would not print that parameter, and the parameter is compared in the default device compare scheme during netlist matching.

Enabling a parameter is rather a hint for the system and the effects can be controlled by other means, so this is not a strong concept. For example, once a tolerance is specified for a parameter, the "primary" flag of the parameter is not considered anymore. The inverse the this function is disable_parameter.

"equivalent_pins" - Marks pins as equivalent

Usage:

- `equivalent_pins(circuit, pin ...)`

This method will mark the given pins as equivalent. This gives the compare algorithm more degrees of freedom when establishing net correspondence. Typically this method is used to declare inputs from gates are equivalent where are are logically, but not physically (e.g. in a CMOS NAND gate):

```
netter.equivalent_pins("NAND2", 0, 1)
```

The circuit argument is either a circuit name (a string) or a Circuit object from the schematic netlist.

Names are case sensitive for layout-derived netlists and case-insensitive for SPICE schematic netlists.

The pin arguments are zero-based pin numbers, where 0 is the first number, 1 the second etc. If the netlist provides named pins, names can be used instead of numbers. Again, use upper case pin names for SPICE netlists.

Use this method andwhere in the script before the compare call.

"ignore_parameter" - Skip a specific parameter for a given device class name during device compare

Usage:

- `ignore_parameter(device_class_name, parameter_name)`

Use this function is ignore a parameter for a particular device class during the netlist compare. Some parameters - for example "L" and "W" parameters of the resistor device - are "secondary" parameters which are not ignored by default. Using "ignore_parameter" on such devices does not have an effect.

"ignore_parameter" and "tolerance" only have an effect with the default device comparer. Using a custom device comparer will override the definitions by "ignore_parameter" or "tolerance".

"join_symmetric_nets" - Joins symmetric nets of selected circuits on the extracted netlist

Usage:

- `join_symmetric_nets(circuit_filter)`

Nets are symmetrical if swapping them would not modify the circuit. Hence they will carry the same potential and can be connected (joined). This will simplify the circuit and can be applied before device combination (e.g. through "netlist.simplify") to render a schematic-equivalent netlist in some cases where symmetric nodes are split (i.e. "split gate" configuration).

This method operates on the extracted netlist (layout). The circuit filter specifies the circuits to which to apply this operation. The filter is a glob-style pattern. Using "*" for all circuits is possible, but it's discouraged currenty until the reliability of the symmetry detection algorithm is established. Currently it is recommended to apply it only to those circuits for which this feature is required.

For the symmetry detection, the specified constraints (e.g. tolerances, device filters etc.) apply.

"lvs_data" - Gets the internal LayoutVsSchematic object

Usage:

- `lvs_data`

The LayoutVsSchematic object provides access to the internal details of the netter object.

"max_branch_complexity" - Configures the maximum branch complexity for ambiguous net matching

Usage:

- `max_branch_complexity(n)`

The netlist compare algorithm is basically a backtracing algorithm. With ambiguous nets, the algorithm picks possible net pairs and tries whether they will make a good match. Following the deduction path for this nets may lead to further branches if more ambiguous nets are encountered. To avoid combinational explosion, the maximum branch complexity is limited to the value configured with this function. The default value is 500 which means not more than 500 combinations are tried for a single seed pair. For networks with inherent ambiguity such as decoders, the complexity can be increased at the expense of potentially larger runtimes. The runtime penality is roughly proportional to the branch complexity.

By default, the branch complexity is unlimited, but it may be reduced in order to limit the compare runtimes at the cost of a less elaborate compare attempt. The preferred solution however is to use labels for net name hints which also reduces the depth.

"max_depth" - Configures the maximum search depth for net match deduction

Usage:

- `max_depth(n)`

The netlist compare algorithm works recursively: once a net equivalence is established, additional matches are derived from this equivalence. Such equivalences in turn are used to derive new equivalences and so on. The maximum depth parameter configures the number of recursions the algorithm performs before picking the next net. With higher values for the depth, the algorithm pursues this "deduction path" in greater depth while with smaller values, the algorithm prefers picking nets in a random fashion as the seeds for this deduction path. The default value is 8.

By default, the depth is unlimited, but it may be reduced in order to limit the compare runtimes at the cost of a less elaborate compare attempt. The preferred solution however is to use labels for net name hints which also reduces the branch complexity.

"max_res" - Ignores resistors with a resistance above a certain value

Usage:

- `max_res(threshold)`

After using this method, the netlist compare will ignore resistor devices with a resistance value above the given threshold (in Farad).

"min_caps" - Ignores capacitors with a capacitance below a certain value

Usage:

- `min_caps(threshold)`

After using this method, the netlist compare will ignore capacitance devices with a capacitance values below the given threshold (in Farad).

"no_lvs_hints" - Disables LVS hints

Usage:

- `no_lvs_hints`

LVS hints may be expensive to compute. Use this function to disable generation of LVS hints

"same_circuits" - Establishes an equivalence between the circuits

Usage:

- `same_circuits(circuit_a, circuit_b)`

This method will force an equivalence between the two circuits. By default, circuits are identified by name. If names are different, this method allows establishing an explicit correspondence.

circuit_a is for the layout netlist, circuit_b for the schematic netlist. Names are case sensitive for layout-derived netlists and case-insensitive for SPICE schematic netlists.

One of the circuits may be nil. In this case, the corresponding other circuit is mapped to "nothing", i.e. ignored.

Use this method andwhere in the script before the compare call.

"same_device_classes" - Establishes an equivalence between the device classes

Usage:

- `same_device_classes(class_a, class_b)`

This method will force an equivalence between the two device classes. Device classes are also known as "models". By default, device classes are identified by name. If names are different, this method allows establishing an explicit correspondence.

Before this method can be used, a schematic netlist needs to be loaded with schematic.

class_a is for the layout netlist, class_b for the schematic netlist. Names are case sensitive for layout-derived netlists and case-insensitive for SPICE schematic netlists.

One of the device classes may be "nil". In this case, the corresponding other device class is mapped to "nothing", i.e. ignored.

A device class on one side can be mapped to multiple other device classes on the other side by using this function multiple times, e.g.

```
same_device_classes("POLYRES", "RES")
same_device_classes("WELLRES", "RES")
```

will match both "POLYRES" and "WELLRES" on the layout side to "RES" on the schematic side.

Once a device class is mentioned with "same_device_classes", matching by name is disabled for this class. So after using 'same_device_classes("A", "B")' "A" is no longer equivalent to "A" on the other side. If you want "A" to stay equivalent to "A" too, you need to use 'same_device_classes("A", "A")' in addition.

Use this method andwhere in the script before the compare call.

"same_nets" - Establishes an equivalence between the nets

Usage:

- `same_nets(circuit_pattern, net_pattern)`
- `same_nets(circuit_pattern, net_a, net_b)`
- `same_nets(circuit_a, net_a, circuit_b, net_b)`

This method will force an equivalence between the net_a and net_b from circuit_a and circuit_b (circuit in the three-argument form is for both circuit_a and circuit_b).

In the four-argument form, the circuits can be either given by name or as Circuit objects. In the three-argument form, the circuits have to be given by name pattern. Nets can be either given by name or as Net objects. In the two-argument form, the circuits and nets have to be given as name pattern.

"name pattern" are glob-style pattern - e.g. the following will identify the all nets starting with "A" from the extracted netlist with the same net from the schematic netlist for all circuits starting with "INV":

```
same_nets("INV*", "A*")
```

After using this function, the compare algorithm will consider these nets equivalent. Use this method to provide hints for the comparer in cases which are difficult to resolve otherwise.

circuit_a and net_a are for the layout netlist, circuit_b and net_b for the schematic netlist. Names are case sensitive for layout-derived netlists and case-insensitive for SPICE schematic netlists.

Use this method andwhere in the script before the compare call.

"same_nets!" - Establishes an equivalence between the nets with matching requirement

Usage:

- `same_nets!(circuit_pattern, net_pattern)`
- `same_nets!(circuit_pattern, net_a, net_b)`
- `same_nets!(circuit_a, net_a, circuit_b, net_b)`

This method is equivalent to same_nets, but requires identity of the given nets. If the specified nets do not match, an error is reported.

"schematic" - Gets, sets or reads the reference netlist

Usage:

- `schematic(filename)`
- `schematic(filename, reader)`
- `schematic(netlist)`
- `schematic`

If no argument is given, the current schematic netlist is returned. nil is returned if no schematic netlist is set yet.

If a filename is given (first two forms), the netlist is read from the given file. If no reader is provided, Spice format will be assumed. The reader object is a NetlistReader object and allows detailed customization of the reader process.

Alternatively, a Netlist object can be given which is obtained from any other source.

"split_gates" - Implements the "split gates" feature

Usage:

- `split_gates(device_name)`
- `split_gates(device_name, circuit_filter)`

Multi-fingered, multi-gate MOS transistors can be built without connecting the source/drain internal nets between the fingers. This will prevent "combine_devices" from combining the single gate transistors of the different fingers into single ones.

"split_gates" now marks the devices of the given class so that they will receive a special treatment which joins the internl source/drain nodes.

By default, this method is applied to all circuits. You can specify a circuit pattern to apply it to certain circuits only.

"device_name" must be a valid device name and denote a MOS3, MOS4, DMOS3 or DMOS4 device.

"tolerance" - Specifies compare tolerances for certain device parameters

Usage:

- `tolerance(device_class_name, parameter_name, absolute_tolerance [, relative_tolerance])`
- `tolerance(device_class_name, parameter_name [, :absolute => absolute_tolerance] [, :relative => relative_tolerance])`

Specifies a compare tolerance for a specific parameter on a given device class. The device class is the name of a device class in the extracted netlist. Tolerances can be given in absolute units or relative or both. The relative tolerance is given as a factor, so 0.1 is a 10% tolerance. Absolute and relative tolerances add, so specifying both allows for a larger deviation.

Some device parameters - like the resistor's "L" and "W" parameters - are not compared by default. These are "secondary" device parameters. Using a tolerance on such parameters will make these parameters being compared even if they are secondary ones.

A function is skip a parameter during the device compare is "ignore_parameter".

"tolerance" and "ignore_parameter" only have an effect with the default device comparer. Using a custom device comparer will override the definitions by "ignore_parameter" or "tolerance".

2.21.2. LVS Reference: Global Functions

Some functions are available on global level and can be used without any object. Most of them are convenience functions that basically act on some default object or provide function-like alternatives for the methods.

LVS is built upon DRC. So all functions available in DRC are also available in LVS. In LVS, DRC functions are used to derive functional layers from original layers or specification of the layout source.

For more details about the DRC functions see DRC::global.

"align" - Aligns the extracted netlist vs. the schematic by flattening circuits where required

Usage:

- `align`

See Netter#align for a description of that function.

"blank_circuit" - Removes the content from the given circuits (blackboxing)

Usage:

- `blank_circuit(circuit_filter)`

See Netter#blank_circuit for a description of that function.

"compare" - Compares the extracted netlist vs. the schematic netlist

Usage:

- `compare`

See Netter#compare for a description of that function.

"consider_net_names" - Indicates whether the netlist comparer shall use net names

Usage:

- `consider_net_names(f)`

See Netter#consider_net_names for a description of that function.

"disable_parameter" - Specifies whether to disable a parameter from a given device class for netlisting and default compare

Usage:

- `disable_parameter(device_class_name, parameter_name)`

See Netter#disable_parameter for a description of that function.

"enable_parameter" - Specifies whether to enable a parameter from a given device class for netlisting and default compare

Usage:

- `enable_parameter(device_class_name, parameter_name)`

See Netter#enable_parameter for a description of that function.

"equivalent_pins" - Marks pins as equivalent

Usage:

- `equivalent_pins(circuit, pins ...)`

See Netter#equivalent_pins for a description of that function.

"ignore_parameter" - Specifies whether to ignore a parameter from a given device class for the compare

Usage:

- `ignore_parameter(device_class_name, parameter_name)`

See Netter#ignore_parameter for a description of that function.

"join_symmetric_nets" - Joins symmetric nets of selected circuits on the extracted netlist

Usage:

- `join_symmetric_nets(circuit_filter)`

See Netter#join_symmetric_nets for a description of that function.

"lvs_data" - Gets the LayoutVsSchematic object after compare was used

Usage:

- `lvs_data`

See Netter#lvs_data for a description of that function.

"max_branch_complexity" - Configures the maximum branch complexity for ambiguous net matching

Usage:

- `max_branch_complexity(n)`

See Netter#max_branch_complexity for a description of that function.

"max_depth" - Configures the maximum search depth for net match deduction

Usage:

- `max_depth(n)`

See Netter#max_depth for a description of that function.

"max_res" - Ignores resistors with a resistance above a certain value

Usage:

- `max_res(threshold)`

See Netter#max_res for a description of that function.

"min_caps" - Ignores capacitors with a capacitance below a certain value

Usage:

- `min_caps(threshold)`

See Netter#min_caps for a description of that function.

"netter" - Creates a new netter object

Usage:

- `netter`

See Netter for more details

"no_lvs_hints" - Disables LVS hints

Usage:

- `no_lvs_hints`

See Netter#no_lvs_hints for a description of that feature.

"report_lvs" - Specifies an LVS report for output

Usage:

- `report_lvs([ filename [, long ] ])`

After the comparison step, the LVS database will be shown in the netlist database browser in a cross-reference view. If a filename is given, the LVS database is also written to this file. If a file name is given and "long" is true, a verbose version of the LVS DB format will be used.

If this method is called together with report_netlist and two files each, two files can be generated - one for the extracted netlist (L2N database) and one for the LVS database. However, report_netlist will only write the extracted netlist while report_lvs will write the LVS database which also includes the extracted netlist.

report_lvs is only effective if a comparison step is included.

"same_circuits" - Establishes an equivalence between the circuits

Usage:

- `same_circuits(circuit_a, circuit_b)`

See Netter#same_circuits for a description of that function.

"same_device_classes" - Establishes an equivalence between the device_classes

Usage:

- `same_device_classes(class_a, class_b)`

See Netter#same_device_classes for a description of that function.

"same_nets" - Establishes an equivalence between the nets

Usage:

- `same_nets(circuit_pattern, net_pattern)`
- `same_nets(circuit_pattern, net_a, net_b)`
- `same_nets(circuit_a, net_a, circuit_b, net_b)`

See Netter#same_nets for a description of that function.

"same_nets!" - Establishes an equivalence between the nets (must match)

Usage:

- `same_nets!(circuit_pattern, net_pattern)`
- `same_nets!(circuit_pattern, net_a, net_b)`
- `same_nets!(circuit_a, net_a, circuit_b, net_b)`

See Netter#same_nets! for a description of that function.

"schematic" - Reads the reference netlist

Usage:

- `schematic(filename)`
- `schematic(filename, reader)`
- `schematic(netlist)`

See Netter#schematic for a description of that function.

"split_gates" - Implements the "split gates" feature for the given device and circuits

Usage:

- `split_gates(device_name)`
- `split_gates(device_name, circuit_filter)`

See Netter#split_gates for a description of that function.

"tolerance" - Specifies compare tolerances for certain device parameters

Usage:

- `tolerance(device_class_name, parameter_name, absolute_tolerance [, relative_tolerance])`
- `tolerance(device_class_name, parameter_name [, :absolute => absolute_tolerance] [, :relative => relative_tolerance])`

See Netter#tolerance for a description of that function.

3. Programming scripts

This category is about programming KLayout using the integrated Ruby or Python scripting language. The following topics are available:

3.1. Introduction

This chapter is about programming extensions for KLayout using the integrated Ruby API (RBA) or Python API (pya).

To use RBA scripts, KLayout must be compiled with the Ruby interpreter. Check under "Help/About" whether support is available. If there is, the "Build options" will include a "Ruby" or "Python" interpreter or both. RBA scripts require a Ruby interpreter. To use pya scripts, Python support must be included.

KLayout comes with the Qt library included into the Ruby or Python API. This means, KLayout scripts can access the full Qt API if Qt binding is available. Check whether "Qt bindings for scripts" is included in the build options on the "Help/About" page.

Basically there are scripts and macros:

- **Scripts** are simple text files which are prepared externally and KLayout acts as an interpreter for these scripts. A special case of scripts is included code, while is loaded into other scripts or macros using "require" on Ruby or "import" on Python. Scripts have been the only way to code Ruby functionality in version 0.21 and earlier.
- **Macros** are special XML files which contain Ruby code plus some additional information required to link them into the system, i.e. automatically execute them on startup or provide menu entries for them. They can load normal ".rb" or ".py" files to implement libraries of classes in the usual way. Macros are managed and developed conveniently in the integrated macro development environment along with the supporting files. This method is the preferred way of creating application extensions and is described in this chapter.

Before you start, please make yourself familiar with the macro development integrated environment (About Macro Development). This documentation also assumes that you familiar with the Ruby programming language. There are numerous books and tutorials about Ruby. The most famous one is the "pickaxe book" (Programming Ruby - The Pragmatic Programmers Guide) by Dave Thomas. If you are familiar with Ruby there is a technical article about the way Ruby and KLayout's core are integrated (The Ruby Language Binding). There are special articles about the integrated Qt binding (The Qt Binding) and PCell programming (Coding PCells In Ruby). If you want to use Python, please read the python implementation article (Using Python) for details about how to translate Ruby samples into Python and specific details of the Python integration.

An introduction into the basic concepts of the KLayout API are given in the article about the application API (The Application API) and about the database API (The Database API).

A First Sample

The first sample is already a complete macro which counts all selected paths, boxes, polygons or text objects. It demonstrates how to set up a macro, how to deal with the selection and how to access the layout database.

Here is the code:

```
module MyMacro

  include RBA

  app = Application.instance
  mw = app.main_window

  lv = mw.current_view
  if lv == nil
    raise "Shape Statistics: No view selected"
  end

  paths = 0
  polygons = 0
  boxes = 0
  texts = 0

  lv.each_object_selected do |sel|

    shape = sel.shape

    if shape.is_path?
      paths += 1
    elsif shape.is_box?
```

```
      boxes += 1
    elsif shape.is_polygon?
      polygons += 1
    elsif shape.is_text?
      texts += 1
    end

  end

  s = "Paths: #{paths}\n"
  s += "Polygons: #{polygons}\n"
  s += "Boxes: #{boxes}\n"
  s += "Texts: #{texts}\n"

  MessageBox::info("Shape Statistics", s, MessageBox::Ok)

end
```

To run the macro, create a new macro in the macro development IDE: choose "Macros/Macro Development". Create a new macro using the "+" button. Rename the macro to a suitable name. Copy the code above into the text. Load a layout, select some objects and in the macro development IDE press F5. A message box will appear the tells us how may boxes, polygons etc. we have selected.

If we look at the code, the first observation is that we put the whole script into our own namespace (we can use any name which is not used otherwise). The advantage of that approach is that we can import the "RBA" namespace which makes life somewhat easier. The RBA namespace contains all KLayout classes and constants. If we would import the RBA namespace into the main namespace we would do that for all other scripts in KLayout since the main namespace is a common resource for all scripts.

The first thing we do inside the macro is to access the layout view:

```
app = Application.instance
mw = app.main_window

lv = mw.current_view
if lv == nil
  raise "Shape Statistics: No view selected"
end
```

The Application class (Application) is a representative for the KLayout application. Since there is only one application, it is a singleton. We can obtain the singleton instance with the class method ("static" in the language of C++) "instance". It delivers a reference to the only Application object which is the main entrance to all internals of KLayout.

The next object which is important is the MainWindow object (MainWindow). Currently there is only one MainWindow object which can be obtained with the "main_window" method of the Application object. The MainWindow object represents the application's window and manages the top level visual objects of the application. The main visual components of the main window are the menus, the tool panels (cell tree, layer list, tool box, navigator ...) and the layout views.

The layout view is the representation of a layout tab (LayoutView). That is basically the window to the layouts loaded into that tab. All related information such as the display settings, the zoom area, the layer properties and the information about the cell shown, the hierarchy levels and further settings go here.

A main window can display multiple tabs. Hence there are multiple LayoutView objects available. The currently selected tab can be addressed with the "current_view" method. This method delivers the LayoutView object associated with that tab. If no layout is loaded, that method returns nil (the Ruby for "nothing")

Actually the preparation step can be simplified without needing the Application and MainWindow object. For demonstration purposes it was included however. Here is the short version:

```
lv = LayoutView.current || raise "Shape Statistics: No view selected"
```

The actual layouts loaded are entities separated from the views. Technically, there is a many-to-many relationship between layout views and layout objects. A layout view may display multiple layouts and one layout may be displayed in multiple layout views. In addition, a layout view can address different cells from a layout. A layout view has a current cell and a path that leads to that cell. The path consists of a specific and unspecific part. The unspecific part of the path tells where the cell we show as the current cell is located in the cell tree. The unspecific part is a tribute to the fact that a cell can appear in different branches of the cell tree (i.e. as child of cell A and cell B). The

specific part of the path addresses a specific instance of within some cell (the "context cell") above in the hierarchy. A specific path is created when we descend down in the hierarchy along a certain instantiation path.

Layout, current cell, context cell, specific and unspecific path are combined into the CellView object (CellView). A layout view can have multiple cell views corresponding to the different layouts that can be loaded into a panel. The cell views do not necessarily have to point to different layouts.

For our sample we don't need the CellView objects, because we get all information directly from the view. But the concept of that object is important to understand the API documentation. Here we ask the layout view for all selected objects and collect the object counts:

```
lv.each_object_selected do |sel|

  shape = sel.shape

  if shape.is_path?
    paths += 1
  elsif shape.is_box?
    boxes += 1
  elsif shape.is_polygon?
    polygons += 1
  elsif shape.is_text?
    texts += 1
  end

end
```

"each_object_selected" is a method of the LayoutView object. More precisely it's an iterator which calls the given block for each selected object. Since the layout view can show multiple layouts, the selected objects may originate from different layouts. In addition, the object may be selected in a child cell of the current cell. Hence, the selection is described by a cell view index (indicating which layout it resided in), an instantiation path (a sequence of instances leading to the cell containing the selected object from the current cell) and the actual object selected. That information is combined into an ObjectInstPath object (ObjectInstPath).

In our case we are not interested in the cell view that shape lives in. Neither are we in the instantiation path. Hence all we need is the shape and we can obtain it with the "shape" method. This method delivers a Shape object (Shape), which is some kind of pointer (a "proxy") to the actual shape. The actual shape is either a polygon, a box, a text or a path. The Shape object has multiple identities and we can ask it what shape type is represents. For that, the Shape object offers methods like "is_box?" etc. If we know the type we can ask it for the actual object and fetch a Polygon (Polygon), a Box (Box), a Text (Text) or a Path object (Path). For our sample however we don't need access to the actual object.

Finally we put together a message and display it in a message box:

```
s = "Paths: #{paths}\n"
s += "Polygons: #{polygons}\n"
s += "Boxes: #{boxes}\n"
s += "Texts: #{texts}\n"

MessageBox::info("Shape Statistics", s, MessageBox::Ok)
```

MessageBox (MessageBox) is a class that provides modal message dialogs through several class methods. "info" shows an information box and with the given title and message. The third parameter indicates which buttons will be shown. In that case, one "Ok" button is sufficient because we don't want to take specific actions when the message box is closed. MessageBox is not the Qt class, which is also available (QMessageBox), but less portable in case a user does not have Qt binding enabled.

This is just a simple example, but it already illustrates some basic concepts. For a in-depth introduction into the API, read The Application API and The Database API.

3.2. Using Python

KLayout does not come with one integrated interpreter. Instead Python and Ruby can **both** be used together. So it is possible to write one script in Ruby and another one in Python. Just pick your favorite language. Scripts written in different languages share the same KLayout data structures. Naturally they cannot directly share variables or language-specific data. But you can, for example, implement PCells in Python and Ruby and use those different PCells in the same layout at the same time. Depending on the type of PCell, KLayout will either execute Python or Ruby code.

Python macros are loaded into KLayout using either ".py" files or ".lym" files with the interpreter set to "Python". To create Python macros, a new tab is available in the Macro Development IDE. When creating macros in the "Python" tab, they will use the Python interpreter. Macros created in the "Ruby" tab will use the Ruby interpreter. Files loaded by "import" need to be in plain text format and use the ".py" suffix. The macro folder is called "pymacros" for a clean separation between the two macro worlds. Technically, both Ruby and Python macros are .lym files with a different interpreter specified in these files.

The Python macro folder is in the "sys.path" search path so it is possible to install modules there. To install libraries globally use "*inst_path*/lib/python/Lib" and "*inst_path*/lib/python/DLLs" on Windows. *inst_path* is the installation path (where klayout.exe is located). On Linux, the installation will share the Python interpreter with the system and modules installed there will be available for KLayout too.

"$PYTHONHOME" is not supported to prevent interference with other Python consumers. Instead, KLayout will read the Python path from "$KLAYOUT_PYTHONPATH" (for Python >= 3.x).

Writing Macros in Python

A good way is to start with the samples provided when creating new macros on the Python tab. The samples are available at the end of the template list. There is a sample for a PCell implementation, a sample for a Qt dialog, a sample for using Qt's .ui files in Python macros and one sample turning KLayout into a HTTP server using a Python macro.

Apart from a few specialities and the different language of course, Python macros do not look much different from Ruby macros. Ruby's "RBA" namespace is "pya" for Python (lowercase to conform with PEP-8). The class and methods names are the same with very few exceptions and the documentation can be used for Python too. Where necessary, a special remark is made regarding the Python implementation.

Here is a basic Python Macro. It creates a layout with a single cell and single layer and puts one rectangle on that layer:

```
# Python version:

import pya

layout = pya.Layout()
top = layout.create_cell("TOP")
l1 = layout.layer(1, 0)
top.shapes(l1).insert(pya.Box(0, 0, 1000, 2000))

layout.write("t.gds")
```

Here is the Ruby variant to demonstrate the similarity:

```
# Ruby version:

layout = RBA::Layout::new()
top = layout.create_cell("TOP")
l1 = layout.layer(1, 0)
top.shapes(l1).insert(RBA::Box::new(0, 0, 1000, 2000))

layout.write("t.gds")
```

Of course, not everything can be translated that easily between Ruby and Python. The details are given below. Usually however, it's straightforward to translate Ruby into Python.

There is no clear advantage of one language over the other. The Python community is somewhat stronger, but performance-wise, Ruby is better. In KLayout, the debugger support for Python is slighly better, since the guts of the interpreter are better documented for Python.

Apart from that, Python and Ruby coexist remarkably well and it is amazing, how easy it it to extend the interfaces from Ruby to Python: not counting the different in the memory management model (mark and sweep garbage collector in Ruby, reference counting in Python), the concepts are very similar.

Please read the Python specific notes below before you start. Some things need to be considered when going from Ruby to Python.

Python PCells

Please have a look at the PCell sample available in the templates. Pick the PCell sample after you have created a new Python macro.

PCell implementation in Python is very similar to Ruby.

Python macros are ".lym" files that are placed into the "pymacro" subfolder in the KLayout path. Python libraries can be put into the "python" subfolder. This subfolder is included into the "sys.path" variable, so macros can load libraries simply by using "import".

Python Implementation Notes

- **KLayout module:** KLayout's module is "pya" (lowercase conforming to PEP 8).
- **Reserved names:** Some methods with reserved names are not available (i.e. "exec", "in"). Some of these methods have been renamed and their original name is still available in Ruby, but use is deprecated. Where this was not possible, they are available with an appended underscore. For example: "QDialog.exec" is available as "QDialog.exec_". That is the same scheme PyQt uses.
- **Assignment methods (attribute setters):** Assignment methods (i.e. "Box#left=" are available as attributes. If there is a read accessor method too, the attribute can be read and written. For example:

  ```
  box = pya.Box()
  box.left = 10
  box.right = box.right + 100
  ```

  If the translation is ambiguous (i.e. because there is more than one getter or setter, the setter will be translated to a method "set_x(value)" where "x" is the attribute name.
- **Predicate getters:** Question-mark names for predicates are translated to non-question-marker names:

  ```
  # Ruby:
    edges.is_empty?

  # Python:
    edges.is_empty()
  ```

- **Constants:** Constants (upper-case static variable) are made available as static attributes.
- **Arrays:** Arrays will be represented as lists, but on assignment, they accept tuples as well.
- **Boolean values:** Boolean values are True and False.
- **No protected methods:** Protected methods are not supported - methods are public always.
- **"nil" value:** The Python equivalent to Ruby's "nil" is "None".
- **Iterators:** Iterator binding:

  ```
  edges = pya.Edges()
  ...
  for edge in edges.each():
    ...
  ```

  If there is an iterator named "each", it will become the default iterator:

```
for edge in edges:
  ...
```

- **Keyword arguments:**
  Most methods support keyword arguments, for example:

  ```
  # a 45 degree rotation
  t = pya.CplxTrans(rot = 45)
  ```

  Exceptions are some built-in methods like "assign". Keyword arguments can be used when the non-optional arguments are specified either as positional or other keyword arguments.

- **Standard protocols:**
  "x.to_s()" is available as "str(x)" too.

  "x.size()" is available as "len(x)" too.

  If there is a "[]" operator and a "size" method, the object implements the sequence protocol too.

- **Operators:**
  Operators are made available through Python operators. For example

- "+" will be available as "+" ("__add__")
- "&" will be available as "and" ("__and__")
- "|" will be available as "or" ("__or__")
- "==" will be available as "==" ("__eq__")

- **Deep copies:** Deep copies of pya objects can be made with dup()

  ```
  box = pya.Box(10, 20, 110, 220)
  copy_box = box.dup()
  ```

- **Events (signals):** Events can be bound to lambdas or functions:

  ```
  action.on_triggered( lambda: action.text += "X" )
  ```

  or to function:

  ```
  def f():
    print "triggered"

  action.on_triggered(f)
  ```

  Events have to match precisely - exactly the number of arguments have to be declared.

- **sys.settrace:** Using "sys.settrace" will disable the debugger support permanently.

- **Instance attributes can't reimplement virtual methods:** This is a limitation driven by the need to avoid cyclic references. Instance-bound methods require a reference to the instance and that will create a cycle with the reimplementation callable object which is held by the class itself.

- **Tips when developing own modules:**
- The "python" subfolders of the KLayout path are added to sys.path, so modules can be put as plain .py files and imported with "import module".
- Or: modules can be put into folders inside "python" using an "__init__.py" file to indicate the folder is a module.
- Use "reload(module)" on the console to refresh the module cache if changes have been applied.

3.3. The Application API

This section covers the basic application API. The application API consists of the main application class and several classes that represent the user interface. This sections presents a selection of classes that make up the application API. These classes provide the main entry points into the application API. Further classes are documented in RBA Class Index.

All classes discussed herein are contained in the RBA namespace. In you code you either have to use qualified names (i.e. RBA::Application) or include the RBA module in you macro's namespace.

The Application class

The Application class is documented in detail in Application. It represents the application and because there is just one application, there also is just one instance of the application object. That instance can be obtained through the "instance" class method:

```
Application::instance
```

The application object is the main entry point into the API. It offers several methods and attributes. In particular:

- Application#application_data_path: returns the user-local storage path. This is where KLayout saves user-specific files, for example the configuration file.
- Application#execute: runs the application. Normally, this method is called implicitly when the application is started. It is possible to use KLayout as a Ruby interpreter by supplying a Ruby script on the command line with the "-r" option. Such scripts must run the application explicitly if they want to.
- Application#exit: exits the application. This method unconditionally terminates the application in a clean way.
- Application#get_config and Application#set_config: read and write the configuration database. The configuration database is a storage of name/value pairs which is stored in the configuration file. These methods can be used to manipulate that storage. Use the Application#get_config_names method to retrieve the names of the configuration parameters stored inside the configuration database. Use the Application#commit_config method to activate settings that have been made with "set_config".
- Application#inst_path: returns the installation path. That is where the executable is located.
- Application#is_editable?: returns true, if KLayout runs in editable mode.
- Application#klayout_path: returns the KLAYOUT_PATH value. This is the search path where KLayout looks for library files or macros. This method delivers the application data path and can be used to look up files required by the macro.
- Application#main_window: delivers the MainWindow object which represents the application's main window. See below for a description of that class.
- Application#process_events: process pending events. If that method is called periodically during long operations, the application will be able to process events and thus handle clicks on a "Stop" button for example. Please note that calling this method is not safe in every context, because not every execution context is reentrant.
- Application#read_config and Application#write_config: reads and writes the configuration database from a file.
- Application#version: delivers KLayout's version string. This string can be used to switch the implementation of a script depending on KLayout's version.

The MainWindow class

The MainWindow class is documented in detail in MainWindow. It represents the main application window. The main window instance can be obtained with:

```
Application::instance.main_window
```

The main window object is the entry point to all user-interface related objects. It offers a couple of methods. In particular:

- MainWindow#cancel: cancels any pending operation (i.e. dragging of an object in move mode) and resets the mode to the default mode (Select). Use this method to establish a known user interface state.
- MainWindow#close_all and MainWindow#close_current_view: close all or the current tab.
- MainWindow#cm_*: these are methods which are bound to the menu items in the menu bar. They can be used to trigger a menu function from a script.
- MainWindow#create_layout: create a new layout and load it into a layout view. This method has a parameter that controls whether the layout is shown in a new tab, replaces the layout in the current tab or adds to the current tab.
- MainWindow#create_layout: creates a new, empty layout and loads it.
- MainWindow#create_view: creates a new, empty tab.
- MainWindow#current_view: returns a LayoutView object (see below) which represents the current tab.
- MainWindow#current_view_index: returns the index of the current tab. Some methods like "select_view" operate with view indexes. The view index is the number of the tab (0 is the leftmost one). The "view" method allows obtaining the LayoutView object from a view index. The same way MainWindow#current_view_index= selects the view with the index given in this call.
- MainWindow#grid_micron: gets the global grid in micrometer units.
- MainWindow#initial_technology and MainWindow#initial_technology=: gets or sets the name of the technology to use to new layouts or for layouts loaded without an explicit technology specification.
- MainWindow#load_layout: loads a layout into a tab. There are various variants of this method, offering various levels of configuration. All these variants have a parameter that controls whether the layout is shown in a new tab, replaces the layout in the current tab or adds to the current tab.
- MainWindow#menu: provides access to the menu object of class AbstractMenu (see AbstractMenu). This object provides access to the main menu, the toolbar and various context menus. With this object it is possible to manipulate the menu.
- MainWindow#message: show a message in the status bar.
- MainWindow#save_session and MainWindow#restore_session: save or restore a session. Sessions contain a window settings and information about the layouts loaded. Sessions allows storing and restoring of the state of the main window.
- MainWindow#select_view: switches to the given tab. This equivalent to using MainWindow#current_view_index=.
- MainWindow#view: gets the LayoutView object for a given tab index.
- MainWindow#views: gets the number of tabs.

The MainWindow supplies three events. See Events And Callbacks for details about events. These are the events:

- **on_current_view_changed**: This event is triggered when the current tab changes. The signal is available with an integer parameter: this is the index of the previous tab. The new tab is already the current tab when this event is triggered.
- **on_view_created**: This event is triggered when a new view is created. The signal is available with an integer parameter: this is the index of the new tab.
- **on_view_closed**: This event is triggered when a new view is closed. The signal is available with an integer parameter: this is the index of the tab that was closed.

In addition, the MainWindow class features many parameterless methods starting with "cm_...". These methods are identical with the methods called when the respective menu functions are triggered. They are of use when menu events need to be emulated in code, for example to implement special key bindings.

The LayoutView class

The LayoutView class is documented in detail in LayoutView. It represents one layout tab in the main window. A single layout view can show multiple layouts. The CellView objects represent one layout loaded into a view. They specific the layout loaded plus the cell selected for drawing. Each LayoutView has a list of CellView objects corresponding to the layouts shown in the same panel.

A LayoutView object can be obtained from the main window either by reading the current view or by getting the view object for a tab by the tab index:

```
# Current view:
Application::instance.main_window.current_view
# or short:
LayoutView::current

# By index:
Application::instance.main_window.view(index)

# Note: the index of the current view is
Application::instance.main_window.current_view_index
# and the number of views is
Application::instance.main_window.views
```

The index is 0 for the first tab. Note that the value returned by LayoutView#current or MainWindow#current_view can be "nil" if no layout is shown.

A layout view is the container for a variety of "visual" objects. These resources are mainly display objects like annotations, markers and images. In addition, the report database system is anchored in the LayoutView object. The following resources are managed in the layout view:

- **Annotations (rulers)**: an arbitrary number of annotations can be registered in the view. The annotations are objects of the Annotation class (Annotation). Annotations are independent of layouts and are defined in micron units.
- **Images**: images are also objects independent of layouts. Any number of images can be placed below the drawn layout.
- **Markers**: markers are temporary overlay objects which can be used a highlights or to add some elaborate annotation to a layout view. Markers can be layout database objects, hence it is possible to draw polygons or other objects over the layout. Markers can be configured to a large degree, so different colors can be used for example. Markers are objects of class Marker (Marker).
- **Local configuration**: be default, the layout view pulls its configuration from the global configuration database. It is possible however to override certain configuration parameters for a particular view. This allows for example to set the background color for a particular view without affecting the other views.
- **Layer properties**: the layer properties tree is also managed by the layout view. Since there can be multiple layer properties trees in different tabs in the layer properties panel, there are method to access either the current or a specific one of the layer properties trees.
- **Custom stipples and line pattern**: custom stipplesu and line styles can be set in the layout view and used in the layer properties. Custom stipples are bitmaps that define the fill pattern used for the inside area of a polygon. Line styles are bit pattern that make lines being resolved into dots.
- **Selection**: the layout view also manages the selection. This is a set of objects and their instantiation path in the layout database. It represents the set of selected objects. Each selected object is described by a instantiation path and the object itself. That information is combined in the ObjectInstPath object (ObjectInstPath).
- **Transient selection**: this is the object that is highlighted briefly when the mouse hovers over it.
- **Cell views**: the list of layouts and cells shown in the layout view as overlays. Cell views are created when layouts are loaded and deleted when layouts are closed. One of the cell views is the "active" one. That is the one which is selected in the drop-down box in the cell tree and for which the cell tree is shown.
- **Cell visibility**: the information about what cell is visible and what cell is not. Each cell can be made invisible. In that case, only the cell frame is drawn and the cell is shown stroked out in the cell tree.

- **Hierarchy levels**: this attribute controls which hierarchy levels are shown by default.
- **Viewport**: the geometrical dimensions of the area which is drawn in micron space.
- **Report databases**: a layout view can have multiple report databases attached to it. Report databases can be shown in the marker browser and are collections of general information or geometrical information related to a certain position or area.
- **Transactions**: transactions are grouped layout operations which form an atomic operation which can be undone. Transactions can be created within the layout view. Transactions must be opened and closed before they are available as operations on the undo stack.
- **Plugins**: plugins are a way to implement new functionality inside the layout view related to mouse actions. By using plugins it is possible to track the mouse and implement actions related to mouse activity.
- **Title**: finally, a layout view has a title which is shown in the tab.

Being the central class, the layout view naturally offers many methods and attributes. Here's a brief explanation of some of these methods:

- LayoutView#active_cellview and LayoutView#active_cellview_index: gets the active CellView object or index of this object. The active cell view is the one that is selected in the drop-down box above the cell tree. CellView#active gets the active cellview of the current layout view.
- LayoutView#add_missing_layers will add the layers to the layer tree for which there is no layer properties entry yet. This method can be used after a layout has been created and populated to show all layers of the layout.
- LayoutView#add_stipple, LayoutView#clear_stipples and LayoutView#remove_stipple: manage custom stipples.
- LayoutView#add_line_style, LayoutView#clear_line_styles and LayoutView#remove_line_style: manage custom line styles.
- LayoutView#ascend and LayoutView#descend: moves the context cell up or down in the hierarchy.
- LayoutView#begin_layers and LayoutView#end_layers: gets an start or end iterator object that allows traversing of the layer properties tree in a recursive or non-recursive fashion. See below for a description of how to deal with the layer properties.
- LayoutView#each_layer is a convenient alternative way of iterating over the layers without directly using the layer tree iterator.
- LayoutView#box: gets the display area in micron units (the viewport).
- LayoutView#cancel: returns the view into idle state (nothing selected, no editing in progress, "Select" mode is active).
- LayoutView#cellview: gets the CellView object for a given index.
- LayoutView#cellviews: gets the number of cell views registered.
- LayoutView#clear_annotations, LayoutView#insert_annotation, LayoutView#erase_annotation and LayoutView#replace_annotation: manage annotations (rulers).
- LayoutView#clear_images, LayoutView#insert_image, LayoutView#erase_image and LayoutView#replace_image: manage images.
- LayoutView#clear_config, LayoutView#get_config, LayoutView#set_config and LayoutView#commit_config: allow manipulation of the configuration for that layout view only. For example it is possible to set a different background color for that specific layout view.
- LayoutView#clear_layers, LayoutView#delete_layer, LayoutView#delete_layer_list, LayoutView#insert_layer, LayoutView#insert_layer_list, LayoutView#replace_layer_node and LayoutView#remove_unused_layers: manage the layer properties. See below for a detailed explanation.
- LayoutView#clear_transactions: clears all transactions (clears the undo stack).
- LayoutView#transaction and LayoutView#commit: starts or ends a transaction. All operations between the start and end of a transaction can be undone in one step.
- LayoutView#clear_object_selection: clears the selection of geometrical objects (shapes or cell instances).

- LayoutView#create_layout and LayoutView#load_layout: creates a new layout or loads a layout. In both cases, you can either replace the current layouts or add the new one to the layouts present.
- LayoutView#current_layer: returns an iterator pointing to the current layer (the one that has the focus frame in the layer tree.
- LayoutView#current_layer_list: returns the index of the current layer list (if multiple tabs are present in the layer control panel, the current layer list is the tab that is selected).
- LayoutView#each_annotation and LayoutView#each_annotation_selected: delivers all or the selected annotations.
- LayoutView#each_image and LayoutView#each_image_selected: delivers all or the selected images.
- LayoutView#each_object_selected and LayoutView#each_object_selected_transient: delivers ObjectInstPath objects (ObjectInstPath) that point to one selected object each.
- LayoutView#erase_cellview: close a cell view, i.e. remove that specific layout from the list of loaded layouts.
- LayoutView#enable_edits: Enables or disables editing. This method will enable or disable all editing features. This is intended for temporarily disallowing edits. This is not the same than edit and viewer mode.
- LayoutView#get_image, LayoutView#get_image_with_options and LayoutView#get_screenshot: dumps the screen content into a QImage with or without a specific resolution.
- LayoutView#init_layer_properties: provides an initialization of a "LayerProperties" object for a new layer according to the current settings of the view.
- LayoutView#is_cell_hidden?, LayoutView#hide_cell, LayoutView#show_cell and LayoutView#show_all_cells: manages cell visibility.
- LayoutView#load_layer_props and LayoutView#save_layer_props: loads or saves layer properties files.
- LayoutView#max_hier, LayoutView#max_hier_levels= and LayoutView#min_hier_levels=: manages hierarchy levels shown.
- LayoutView#object_selection, LayoutView#object_selection=, LayoutView#select_object and LayoutView#clear_object_selection: return or manipulate the selection of geometrical objects (shapes, instances). The key descriptor object for that purpose is ObjectInstPath, which refers to a geometrical object throug an instantiation path.
- LayoutView#pan_center (and other pan... methods), LayoutView#zoom_box (and other zoom... methods): changes the viewport.
- LayoutView#reload_layout: reloads a given layout.
- LayoutView#create_rdb, LayoutView#remove_rdb, LayoutView#rdb and LayoutView#num_rdbs: Create, delete and get report databases stored inside the LayoutView object.
- LayoutView#rename_cellview: changes the name of a cellview.
- LayoutView#title, LayoutView#title= and LayoutView#reset_title: sets or resets the layout view's title.
- LayoutView#save_as: Saves a layout to a file (with options).
- LayoutView#show_image: shows or hides an image.
- LayoutView#viewport_width, LayoutView#viewport_height and LayoutView#viewport_trans: gets the viewport parameters.

Implementing Undo/Redo

Undo/Redo functionality is implemented by using "transactions". Transactions are groups of operations which implement one user operation. Transactions are built internally and automatically once a transaction is initiated. Most operations performed in the framework of the LayoutView and Layout objects are tracked within these transactions. When a transacting is finished, it needs to be committed. After that, a new operation will be available for "Undo" or "Redo".

Transactions can be initiated with LayoutView#transaction and committed with LayoutView#commit. To ensure, every initiation of a transaction is matched by a "commit", it is recommended to employ "ensure":

```
begin

  view.transaction("Some operation")

  ... do your thing here ...

ensure
  view.commit
end
```

Manipulating the selection

The selection of geometrical objects can be manipulated by providing the necessary ObjectInstPath objects. Each such object provides a "pointer" to a shape or instance through the hierarchy. Specifically it lists all the cells and their instantiation transformations down to the shape selected. By accumulating these selections, a shape can be addressed in a flat view, even if the shape is instantiated many levels down in the hierarchy.

Generating such instantiation path objects is somewhat tedious, but usually the requirement is not to generate such paths, but to take an existing selecting, manipulate it somehow and then to set the selection to the new one. This is fairly easy by taking a copy of the selection, manipulation of the shapes and setting the manipulated selection as the new one.

The following is a sample which replaces all shapes by their hull polygons. Note that is provides undo/redo support through a "transaction":

```
view = mw.current_view

begin

  view.transaction("Convert selected shapes to polygons")

  sel = view.object_selection

  sel.each do |s|
    if !s.is_cell_inst? && !s.shape.is_text?
      ly = view.cellview(s.cv_index).layout
      # convert to polygon
      s.shape.polygon = s.shape.polygon
    end
  end

  view.object_selection = sel

ensure
  view.commit
end
```

Events

The LayoutView object supplies several events. See Events And Callbacks for details about events. These are the events:

- **on_active_cellview_changed**: This event is triggered when the active cellview changes. The active cellview is the one indicated by the drop-down-box atop of the cell list if multiple layouts are loaded into one view.
- **on_annotation_changed**: This event is triggered if an annotation is changed. The ID of the annotation is sent along with the event.
- **on_annotation_selection_changed**: This event is triggered if the selection of annotations is changed.
- **on_annotations_changed**: This event is triggered if an annotation is added or deleted.
- **on_cell_visibility_changed**: This event is triggered when the visibility of a cell changes. The visibility of a cell is changed by using "Hide Cell" or "Show Cell" from the cell tree's context menu.
- **on_cellviews_changed**: This event is triggered when a new cellview is added or a cellview is removed.

- **on_cellview_changed**: This event is triggered when a cellview changed (i.e. the current cell has been changed. The index of the changed cell view is sent along with the event.
- **on_close**: This event is triggered when a cell view is closed.
- **on_file_open**: This event is triggered when a file is loaded.
- **on_hide**: This event is triggered when a cell view is going to become invisible (i.e the tab changed).
- **on_current_layer_list_changed**: This event is triggered when the current layer list was changed (i.e. the tab in the layer list has been changed).
- **on_image_changed**: This event is triggered when an image was edited. The ID of the image is sent along with the event.
- **on_image_selection_changed**: This event is triggered when an image was selected or unselected.
- **on_images_changed**: This event is triggered when an image was added or deleted.
- **on_layer_list_changed**: This event is triggered if a layer was changed, added or deleted.
- **on_layer_list_deleted**: This event is triggered if a layer list was deleted (i.e. a tab was removed).
- **on_layer_list_inserted**: This event is triggered if a layer list was inserted (i.e. a tab was added).
- **on_rdb_list_changed**: This event is triggered when a report database is opened or removed.
- **on_selection_changed**: This event is triggered when the selection has changed.
- **on_show**: This event is triggered when a cell view is going to become visible (i.e the tab changed).
- **on_transient_selection_changed**: This event is triggered when the transient selection has changed.
- **on_viewport_changed**: This event is triggered when the viewport has changed, for example the view is zoomed in or panned.

Working with layer properties

The API provides methods by which the layer properties list of the layout view can be traversed and manipulated in many ways. In particular:

- Add or remove entries to or from the layer properties list: LayoutView#insert_layer and LayoutView#delete_layer
- Clear all entries: LayoutView#clear_layers
- Manage custom stipple pattern: LayoutView#add_stipple, LayoutView#clear_stipples and LayoutView#remove_stipple
- Add, rename or remove tabs, get or change the current tab: LayoutView#insert_layer_list, LayoutView#rename_layer_list, LayoutView#delete_layer_list, LayoutView#current_layer_list and LayoutView#set_current_layer_list
- Remove unused layers from the layer list or add entries for missing layers: LayoutView#remove_unused_layers, LayoutView#add_missing_layers
- Load or save layer properties from or to a ".lyp" file: LayoutView#load_layer_props and LayoutView#save_layer_props
- Obtain the selected entries from the layer properties tree: LayoutView#selected_layers and LayoutView#current_layer
- Initialize layer properties with the default settings: LayoutView#init_layer_properties
- Expand layer properties which contain wildcard entries ("stylesheet" layer properties files): LayoutView#expand_layer_properties
- Manipulate layer properties by setting the properties of the LayerPropertiesNodeRef object returned by LayoutView#each_layer or LayerPropertiesIterator#current.

Many of these functions use LayerPropertiesIterator objects to identify entries in the layer tree. Such an object is basically a pointer into the tree. The term "iterator" refers means that such a pointer can be moved to neighboring entries in the layer tree. By default, the LayerPropertiesIterator performs a preorder, depth-first traversal of the layer properties tree (the virtual root object is omitted). This is how to work with LayerPropertiesIterator objects:

```
layout_view = Application::instance.main_window.current_view
# Get the iterator for the first entry:
lp = layout_view.begin_layers
# advance to the next entry (preorder, depth-first traversal):
lp.next
# advance to the next sibling
lp.next_sibling(1)
# advance to the previous sibling
lp.next_sibling(-1)
# move down in the hierarchy to the first child
lp.down_first_child
# move down in the hierarchy to the last child
lp.down_last_child
# move up to the parent node
lp.up
# get the value of the current node
props = lp.current
```

The LayerPropertiesIterator has a couple of attributes:

- LayerPropertiesIterator#current: returns the LayerPropertiesNodeRef object (LayerPropertiesNodeRef) which is a representative for the layer the iterator points to. This object can be manipulated which has an immediate effect on the layer list of the view.
- LayerPropertiesIterator#at_end?: returns true, if the iterator is at the end of the layer properties tree (if "next" is used to traverse) or at the end of the current child node list if other methods of traversal are used. It "at_end?" is true, the iterator does not point to a valid entry.
- LayerPropertiesIterator#is_null?: returns true, if this iterator is a null iterator (i.e. default-constructed).
- LayerPropertiesIterator#at_top?: returns true, if this iterator is pointing to a top-level entry.
- LayerPropertiesIterator#child_index: returns the index of the current entry in the current child list.

Iterators can be compared against each other. If two iterators point to the same object, the equality operator "==" returns true.

The actual entry that the iterators "current" property is a LayerPropertiesNodeRef object (a reference to a LayerPropertiesNode object). If behaves the same way than a LayerPropertiesNode object (LayerPropertiesNode), but modifications of the latter will change the way the layer is displayed in the view.

The LayerPropertiesNode object contributes only a few methods, namely:

- LayerPropertiesNode#id: an integer ID that uniquely identifies the entry in the tree.
- LayerPropertiesNode#flat: computes and delivers an effective set of properties as a LayerProperties object.
- LayerPropertiesNode#bbox: computes the bounding box of the drawn layer represented by this entry.
- LayerPropertiesNode#has_children?: returns true, if this node is not a leaf node.
- LayerPropertiesNode#add_child: adds a child node to the node. It returns a reference to the new node created inside the node's hierarchy. Is is possible to add new children to the node returned.
- LayerPropertiesNode#clear_children: removes all children from the node.

The actual properties of the layer are accessible through methods of the LayerProperties object. Since the parent node may override or contribute properties, a LayerProperties object has a twofold identity: the way it appears finally ("real") and the way it is configured ("local"). The property accessors have a "real" parameter and deliver the real value if this parameter is set to true and the local value otherwise. There are also convenience methods which always deliver the "real" value.

```
lp = layout_view.begin_layers

# manipulate the layer
lp.current.width = 2
lp.current.fill_color = 0x80ff40

# which is equivalent to this somewhat more efficient way:
props = lp.current.dup
props.width = 2
props.fill_color = 0x80ff40
lp.current.assign(props)
```

It is possible to directly manipulate the hierarchy this way:

```
lp = layout_view.begin_layers
# create a copy that we can manipulate
# add two child nodes
cp = RBA::LayerProperties::new
cp.source = "100/0"
lp.current.add_child(cp)
cp = RBA::LayerProperties::new
cp.source = "101/0"
lp.current.add_child(cp)
```

New entries can be created by using LayoutView's insert_layer method and a LayerPropertiesIterator to specify the location where the node shall be created. Here is an example how to create a child entry using that technique. Please note how "down_first_child" is used to navigate into the node's child space which works even if there are no children yet:

```
lp = layout_view.begin_layers
# let the iterator point to the first child, even if it does not exist
lp.down_first_child
# (lp.current may not be valid, but still lp is a valid insert position)
# prepare a new entry for insert:
props = RBA::LayerProperties.new
props.source = "100/0"
# insert the child node:
layout_view.insert_layer(lp, props)
# now, lp points to a valid object: lp.current.source == "100/0"
```

LayerProperties objects

The LayerProperties object represents one entry in the layer properties tree and has several basic properties. For each of these properties, a getter for the real and local value exists as well as a setter that installs a local value. For example, for the width property, the following methods are defined:

- **width(real)**: the getter for the real ("width(true)") or local ("width(false)") value.
- **width**: the real value.
- **width=**: the setter for the local value.

Width is a "weak" property. That means that for computing the effective width, child nodes can override the settings inherited from the parent nodes. A width of 0 is considered "not set" and does not override parent defined widths. Other properties like visibility are "strong", i.e. the parent can override the properties set for its children. Another form of combination is "additive" where the effective property value is the "sum" (or in general combination) of all local properties from parent to child.

Some properties like "fill_color" do not have a neutral value but instead they can be cleared (in that case with "clear_fill_color"). The LayerProperties object can be asked whether a fill color is set using the "has_fill_color?" method.

This is a brief list of properties:

- LayerProperties#animation (strong): specifies animation (blinking, scrolling)
- LayerProperties#dither_pattern (strong): specifies the fill pattern
- LayerProperties#line_style (strong): specifies the line style
- LayerProperties#fill_brightness (additive): specifies the fill color's brightness
- LayerProperties#fill_color (strong): specifies the fill color
- LayerProperties#frame_brightness (additive): specifies the frame color's brightness
- LayerProperties#frame_color (strong): specifies the frame color
- LayerProperties#lower_hier_level (weak): the lower hierarchy level shown. This property has various flavors related to the definition of "lower level".
- LayerProperties#upper_hier_level (weak): the upper hierarchy level shown. This property has various flavors related to the definition of "upper level".
- LayerProperties#marked? (strong): specifies whether the layout is rendered with small crosses at each vertex
- LayerProperties#xfill? (strong): Specifies whether a cross is drawn in rectangles
- LayerProperties#transparent? (strong): specifies whether the layer is semi-transparent (color bitmap combination)
- LayerProperties#visible? (strong): specifies whether the layer is visible
- LayerProperties#valid? (strong): specifies whether the layer is valid
- LayerProperties#width (weak): specifies the line width
- LayerProperties#source (weak, additive): specifies the origin of the data. This property can be set or obtained either a string using KLayout's source notation or be accessed through a couple of specialized properties delivering a part of the source specification each (LayerProperties#source_layer, LayerProperties#source_datatype, LayerProperties#source_name, LayerProperties#source_layer_index, LayerProperties#source_cellview and LayerProperties#trans)

In addition, a couple of getters for computed and derived values are present (i.e. "eff_frame_color"). There are no setters for these properties. The effective frame color for example delivers the frame color which results from combining the frame color and the frame brightness.

The CellView class

The CellView (CellView) identifies the cell drawn and the context the cell is drawn in. A CellView can be created as a object but usually it is obtained from a LayoutView object. In the following example, the active cell view is used:

```
RBA::Application::instance.main_window.current_view.active_cellview
```

Alternatively, a cell view can be addressed by index:

```
lv = RBA::Application::instance.main_window.current_view
num_cellviews = lv.cellviews # number of cell views
lv.cellview(0) # first one
```

A cellview carries the following information:

- CellView#cell: a reference to the cell shown (a Cell object: Cell).
- CellView#layout: a reference to the layout object (a Layout object: Layout) which contains the cell shown.

- CellView#name: the unique name of the layout.
- CellView#filename: the name (actually the path) of the file loaded if the layout was loaded from a file.
- CellView#path: the unspecific path (see below)
- CellView#context_path: the specific path (see below)
- CellView#ctx_cell: the context cell (see below). Alternatively the cell index of the context cell is available by CellView#ctx_cell_index.
- CellView#technology: the technology (name) used with this layout
- CellView#is_cell_hidden?: returns a value indicating whether a given cell is hidden

A cellview can be manipulated to change the cell shown in the layout view. For this purpose, assignment methods exist which will reconfigure the cellview:

- CellView#cell=: a reference to the cell shown (a Cell object: Cell).
- CellView#cell_index=: a reference to the cell shown (by cell index).
- CellView#name=: the unique name of the layout.
- CellView#path=: sets the unspecific path (see below)
- CellView#context_path=: sets the specific path (see below)
- CellView#technology=: applies the given technology to this layout
- CellView#show_cell, CellView#show_all_cells, CellView#hide_cell: shows or hides cells
- CellView#close: closes this cell view (removes it from the layout view)

Unspecific and specific path and context cell

In addition to the cell itself, the cell view specifies how the cell is embedded in the hierarchy. Embedding can happen in two ways: an unspecific and a specific way. Both ways contribute to a path which leads from a top cell to the cell drawn.

The first part is always the unspecific path. This path specifies, where the cell drawn is located in the cell tree. That has no effect on the drawing, but is determines what entry in the cell tree is selected. Giving a path for that information is required, because a cell can be child of different cells which itself can be children of other cells. The unspecific path lists the top cell and further cells which are all direct or indirect parents of the cell addressed.

The unspecific path ends at the "context cell" which usually is identical to the cell addressed by the cell view. KLayout allows addressing of a specific instance of a direct or indirect child cell as the actual cell. In that case, the specific path comes into play. Bascially that means, that a cell is drawn within a context of embedding layout. The specific path leads from the context cell to the cell view's target cell and consists of specific instances (hence the name "specific path"). The "descend" and "ascend" feature bascially adds or removes instances from that path.

The unspecific path can be obtained with the CellView#path method, the specific path with the CellView#context_path method. The unspecific path is just an array of cell indexes specifying the top cell and further cells down to the context cell and includes the context cell. The specific path is an array of InstElement objects (InstElement). Each InstElement object describes a specific instantiation (a cell instance plus information when a specific array instance is addressed). When there is no context, the specific path is an empty array. Using the setters CellView#path= and CellView#context_path= these paths can be changed to select a new cell into the layout view.

The Image class

Images can be placed onto the drawing canvas and display colored or monochrome images below the layout. Images are represented by Image objects (Image). Basically an image is a two-dimensional array of pixel values with a specification how these pixels are to be displayed on the canvas. An image can be created an placed on the canvas like this:

```
lv = RBA::Application::instance.main_window.current_view
image = RBA::Image::new("image.png")
```

```
lv.insert_image(image)
```

An image can be configured by using different properties and attributes:

- The images' data can be loaded from a file by using a constructor with a file name. In addition, the image can use data from an array of floating-point values using either a constructor or the Image#set_data method. An image can be colored, in which case three channels are present or it can be monochrome. In the latter case, a single channel is present only. Together with the data, the dimensions of the image have to be specified (width and height in pixel units).
- The image's data can be manipulated per pixel using the Image#get_pixel or Image#set_pixel method.
- The data range for the data stored in the image can be set using the Image#min_value= and Image#max_value= attributes. The data range determines which value is considered "maximum intensity" (max_value) and "zero intensity" (min_value).
- For monochrome images, a data mapping can be specified. A data mapping converts a monochrome value (a scalar) to a color. Data mapping is specified through a ImageDataMapping object (ImageDataMapping) using the Image#data_mapping method.
- The geometrical properties of an image are encapsulated in a Matrix3d object (Matrix3d). Such a matrix describes the transformation from pixel coordinates to the micron unit space of the canvas. A 3x3 matrix is a generic way to specify a transformation, including translation, rotation, mirror, shear or perspective distortion. The matrix is obtained and set using the Image#matrix attribute. Convenience methods like Image#trans, Image#pixel_width and Image#pixel_height allow accessing sub-aspects of the generic transformation (affine transformation, scaling).

An image can be transformed using one of the Image#transformed methods. It can be hidden or shown using the Image#visible= method. The bounding box of the image can be obtained with the Image#box method.

The Annotation class

Annotations (Annotation) are basically rulers and other "overlay objects" but can be used for other purposes as well, for example to simply add a text object. Annotations, like images, are objects stored in the LayoutView and can be selected, deleted, transformed etc.

Programmatically, annotations are created this way:

```
lv = RBA::Application::instance.main_window.current_view
ant = RBA::Annotation::new
ant.p1 = RBA::DPoint.new(0.0, 0.0)
ant.p2 = RBA::DPoint.new(100.0, 0.0)
lv.insert_annotation(ant)
```

The annotation carries several attributes. Those are the same attributes that can be configured in the annotation properties dialog. The most important properties are the two positions (start and end position) accessible through the Annotation#p1 and Annotation#p2 properties, the style (Annotation#style property) and the outline (Annotation#outline property).

If properties are changed using the attribute setters, their appearance will change as well. The following example demonstrates how rulers are manipulated. In this example, the style of all rulers is set to "arrow on both sides". Note, how in this example transactions are used to implement undo/redo:

```
view = RBA::LayoutView::current

begin

  view.transaction("Restyle annotations")

  view.each_annotation do |a|
    a.style = RBA::Annotation::StyleArrowBoth
  end

ensure
  view.commit
end
```

The Marker class

A marker is a temporary highlight object. A marker is represented by the Marker class (Marker). Markers appear when they are created and disappear when they are destroyed. Since destruction by the garbage collector happens at undefined times, the destroy method can be used to destroy the marker explicitly. Markers accept some plain shapes (i.e. a Box) which will be displayed as the marker. Markers can be configures in manifold ways, i.e. the colors, the fill pattern, line width etc. See the class documentation for details about the configuration properties.

This is how to create and destroy a marker:

```
lv = RBA::Application::instance.main_window.current_view
marker = RBA::Marker.new(lv)
marker.set(RBA::DBox::new(0.0, 0.0, 100.0, 200.0))
# to hide the marker:
marker.destroy
```

Markers are temporary objects intended for highlighting a certain area or shape. Markers are not persisted in sessions nor can they be edited.

The Plugin and PluginFactory classes

Plugins (Plugin) are objects which provide modular extensions of KLayout. Plugins are the only way to handle mouse events in the canvas. The basic operation of a plugin is the following:

- For each plugin type, a PluginFactory (PluginFactory) object must be provided. KLayout uses this object to configure itself and to create a particular plugin instance for each LayoutView. The PluginFactory must provide certain configuration information and can handle some events in a global manner, for example menu entries that do not refer to a certain plugin instance. The PluginFactory must register itself in the KLayout framework. After doing so, KLayout will provide a new button in the tool bar. If this button is selected, the plugin will be activated.
- When a LayoutView is created, it will use the PluginFactory to create a specific Plugin instance for the view. When the tool bar button is pressed which relates to this plugin, the plugin will be activated and mouse or other events will be redirected to this plugin.

The PluginFactory itself acts as a singleton per plugin class and provides not only the ability to create Plugin objects but also a couple of configuration options and a global handler for configuration and menu events. The configuration includes:

- Menu items: by configuring menu items in the PluginFactory, KLayout can create these items when the plugin is initialized. Each menu entry is connected with the plugin through a symbol: this is a string that tells the plugin's Plugin#menu_activated method which menu item was selected. By configuring a menu rather than creating it explicitly, KLayout has a somewhat better control over what menu items belong to which plugin. Menu items are configured by calling PluginFactory#add_menu_entry in the PluginFactory's constructor.
- Configuration options: Instead of directly taking values from the configuration database, it is more convenient to register configuration keys in the PluginFactory's constructor using the PluginFactory#add_option method. After an option is configured, the individual Plugin objects and the PluginFactory receives "configure" calls when a configuration option changes or for the initial configuration.

A PluginFactory must be instantiated and register itself. Menu items and configuration options should be set before the object is registered. Upon registration, a unique name must be specified for the plugin class. Also, the tool button title and optionally an icon can be specified.

The main objective of the PluginFactory class however is to create the actual plugin object. For this, the create_plugin method needs to be reimplemented. The implementation is supposed to create an object of the specific class.

The actual implementation of the plugin is a class derived from Plugin (Plugin). The plugin comes into life, when it is activated. That is, when the tool button is pressed that is associated with the plugin. When the plugin is activated, the Plugin#activated method is called. The method can be reimplemented in order to prepare the plugin for taking actions on mouse events. When the plugin is not longer active, i.e. because another mode has been selected, the Plugin#deactivated method is called.

Every plugin has the ability to receive and intercept mouse events. Various mouse events are available: mouse moved, mouse button clicked (button pressed and released), mouse button double clicked, mouse button pressed, mouse button released, entry or leave of the window and agitation of the mouse wheel. Each event follows a certain protocol depending whether the plugin is active or not. In addition, plugins can request exclusive control over the mouse by "grabbing" the mouse. Each event is associated with a certain callback. The callback has a parameter - "prio" - which determines the role of the event. The protocol is described here:

- First, all plugins that grabbed the mouse with grab_mouse will receive an event callback with 'prio' set to true in the reverse order the plugins grabbed the mouse. If any one of the mouse event handlers returns true, the protocol terminates.
- If that is not the case or no plugin has grabbed the mouse, the active plugin receives the mouse event with 'prio' set to true.
- If no receiver accepted the mouse event by returning true, it is sent again to all plugins with 'prio' set to false. Again, the loop terminates if one of the receivers returns true. The second pass gives inactive plugins a chance to monitor the mouse and implement specific actions - i.e. displaying the current position.

In an mouse event handler, the plugin can take any action, i.e. transform objects or create/remove markers. This allows implementing of interactive functionality upon KLayout's canvas object. Using "set_cursor", the plugin can set the mouse cursor to a specific shape for example. A plugin should consider implementing "drag_cancel" in order to terminate any pending dragging operations. Plugin#drag_cancel is called by KLayout to regain control over the mouse in certain circumstances and is supposed to put the plugin into a "watching" instead of "dragging" state.

3.4. The Database API

The basic object of the database is the Layout object. A Layout object represents a layout file or a layout generated. Basically, a layout is a collection of cells and geometrical shapes. The shapes are organised in layers and the cells can be instantiated in other cells creating a hierarchy of cells. Hence, the basic objects of the database are Layout, Cell, Shapes and the geometrical primitives like Box, Path, Polygon, Text and Edge.

The Layout class

The Layout object is the basic container for a layout. Multiple layouts can live within KLayout. Some are stored inside the application and are created when a layout is loaded for example. However, layout objects can also be created as standalone objects for manipulation by Ruby scripts. Such a layout can be an isolated entity without connection to a view.

A basic sample

This is a sample how to create a layout in a layout view with one cell ("TOP"), one layer (layer 10, datatype 0) and one shape (a 1x2 micron box). This sample also shows how to set up the layout view properly:

```
# create a new view (mode 1) with an empty layout
main_window = RBA::Application::instance.main_window
layout = main_window.create_layout(1).layout
layout_view = main_window.current_view

# set the database unit (shown as an example, the default is 0.001)
layout.dbu = 0.001

# create a cell
cell = layout.create_cell("TOP")

# create a layer
layer_index = layout.insert_layer(RBA::LayerInfo::new(10, 0))

# add a shape
cell.shapes(layer_index).insert(RBA::Box::new(0, 0, 1000, 2000))

# select the top cell in the view, set up the view's layer list and
# fit the viewport to the extensions of our layout
layout_view.select_cell(cell.cell_index, 0)
layout_view.add_missing_layers
layout_view.zoom_fit
```

It is also possible to create a standalone layout. Here is an example how to create the layout without a connection to a view and save that layout to a file:

```
# create the layout
layout = RBA::Layout::new

# set the database unit (shown as an example, the default is 0.001)
layout.dbu = 0.001

# create a cell
cell = layout.create_cell("TOP")

# create a layer
layer_index = layout.insert_layer(RBA::LayerInfo::new(10, 0))

# add a shape
cell.shapes(layer_index).insert(RBA::Box::new(0, 0, 1000, 2000))

# save the layout
layout.write("my_layout.gds")
```

Overview over the Layout object

The basic building blocks of layouts are layers and cells. Layers are not individual objects. Instead, a layer is rather an index and the various methods allow addressing shapes inside layers by using that index. However, the layout object stores the layer properties, i.e. the layer and datatype number.

Cells are represented by Cell objects which are described later. Cells are often referred to by a cell index which is basically an ID which allows identification of a cell without having to keep a reference. The Layout object allows converting of a cell index to a cell reference with the Layout#cell method. Cells can be instantiated inside other cells. Instances are described by CellInstArray objects.

Another important property of the layout object is the database unit. In the layout, all geometrical coordinates are stored as integer values for efficiency. The database unit specifies the length of one unit in micrometers. The database unit can be accessed with the Layout#dbu attribute. Changing the database unit effectively scales a layout. Often the database unit is a fixed value and compatibility between different layouts in a flow often demands use of a specific database unit. Hence, changing the value of the database unit is possible but requires some careful consideration.

The Layout object keeps shapes (texts, polygons, boxes, paths etc.) on "layers". A layer is a collection of shapes. A layout features a set of layers and each cell provides space for each layer. The shapes are stored inside the cells while the layers are managed by the layout. For doing so, the Layout object keeps layers as a table of LayerInfo objects. The LayerInfo object carries information about the description of a layer, for example layer and datatype number and/or the layer name. A layer is basically just an index in that table. Layers can be created using the Layout#insert_layer method. Each layer is present in every cell and inside a cell, a shape storage for each layer is provided.

Inside the layout object, shapes are kept with integer coordinates. The physical units can be obtained by multiplying the integer coordinates with the database unit. The integer type objects are Box, Polygon etc. Most objects also support floating-point coordinate objects (DBox, DPolygon etc.). These objects are then given in micrometer units - i.e. already multiplied by the database unit. If using these objects, keep in mind that internally they are still integer-type objects. This means, rounding to the database will happen and if you change the database unit, the objects will effectively scale.

A Layout object provides some basic layout manipulation and query methods. For example, it provides a method to retrieve shapes touching or overlapping a certain rectangular region of a cell. It also provides a clip method which extracts a rectangular region from a layout (Layout#clip, Layout#clip_into, Layout#multi_clip, Layout#multi_clip_into). It provides methods to delete cells and cell trees and (Layout#delete_cell, Layout#delete_cell_rec, Layout#delete_cells, Layout#prune_cell, Layout#prune_subcells). There are also methods to manipulate layers (Layout#clear_layer, Layout#copy_layer, Layout#move_layer, Layout#delete_layer).

Some convenience functions are provide to read and write a layout from or to a file. The Layout#read method reads a layout from a file. It basically merges the contents of the file with the layout so it's possible to combine multiple files by using read more than once. The method comes in two flavors: a simple one and one that allows specification of reader options with a LoadLayoutOptions object. There is also a Layout#write method which writes the layout to a file. The simple form writes the layout to a file and the file type is determined by the file extension. A full-featured version exists which allows to specify the format and many more options with a SaveLayoutOptions object.

Layouts can also import cells from Libraries (Library). Such imported cells are basically cells linked to another cell inside the library. Library cells are imported using the Layout#add_lib_cell method. This method creates a "proxy" cell which is a copy of the library cell but is linked to the library. As long as the library is present in the system, this link is maintained and stored in the layout files. If the link is lost because the library is removed, the proxy cell becomes a normal cell. Such proxy cells basically behave like normal cells but should not be manipulated.

Layout objects are also responsible for handling properties. Properties are basically arbitrary sets of data (key/value pairs) attached to shapes, cells or instances. For efficiency, the property data is not attached to every shape, cell or instance. Instead, the layout object manages different property sets and associate each distinct set with an integer ID. The shape, cell or cell instance only stores that ID. To create, query or change property sets, the layout object provides the Layout#properties and Layout#properties_id methods. Since that is inconvenient, shapes, cells and instances provide access to the properties by providing methods to set, get and delete properties from the set (for example Shape#property, Shape#delete_property and Shape#set_property). Internally, these methods create a new ID if necessary and assign that ID to the shape, cell or instance.

A layout can provide and import PCells. PCells are cells that provide its geometry through program code (for example written in Ruby) and provide parameters which can be adjusted to change the appearance of the cell. For each PCell a "declaration" must be provided which basically contains the code for the PCell and some information about the parameters provided by the PCell. PCells are stored in the layout and are referred to by a PCell ID (an integer). PCells are added to a layout using Layout#register_pcell and retrieved by ID or name using Layout#pcell_declaration. PCells are instantiated with a specific parameter set using the Layout#add_pcell_variant. This method creates a cell representing the layout generated by the PCell code for a particular set of parameters. The layout internally caches the PCell layouts so the PCell code is executed only if a new parameter set is requested. Usually PCells are provided through libraries. In that case, the library provides the PCell variant through Layout#add_pcell_variant which is imported into the target layout through Layout#add_lib_cell. There is a overload of Layout#add_pcell_variant which combines both steps.

The following code demonstrates how to create a PCell (in that case a "TEXT" cell from the "Basic" library):

```
ly = RBA::Layout.new
top = ly.add_cell("TOP")

# Find the lib
lib = RBA::Library.library_by_name("Basic")
lib || raise("Unknown lib 'Basic'")

# Find the pcell
pcell_decl = lib.layout.pcell_declaration("TEXT")
pcell_decl || raise("Unknown PCell 'TEXT'")

# Set the parameters (text string, layer to 10/0, magnification to 2.5)
param = { "text" => "KLAYOUT RULES", "layer" => RBA::LayerInfo::new(10, 0), "mag" => 2.5 }

# Build a param array using the param hash as a source.
# Fill all remaining parameter with default values.
pv = pcell_decl.get_parameters.collect do |p|
  param[p.name] || p.default
end

# Create a PCell variant cell
pcell_var = ly.add_pcell_variant(lib, pcell_decl.id, pv)

# Instantiate that cell
t = RBA::Trans::new(RBA::Trans::r90, 0, 0)
pcell_inst = ly.cell(top).insert(RBA::CellInstArray::new(pcell_var, t))
```

Editable mode

A layout can exist in two flavors: editable and non-editable. In editable mode, some optimisations are disabled. For example, OASIS shape arrays are expanded into single shapes. This enables manipulation of the database. For example, the shape replacement, property manipulation and other operations are only possible in editable mode. On the other hand, the memory footprint of a layout may be larger in editable mode. Independent of the mode, layouts can be created and cells, instances and shapes can be added, but not manipulated.

A layout object living in the application is created in editable or non-editable mode depending on the application setting. Layout objects explicitly created by RBA code can either be in editable or non-editable mode:

```
editable_layout = RBA::Layout.new
non_editable_layout = RBA::Layout.new(false)
```

The Layout#is_editable? method returns true, if a layout is in editable mode. Once the layout is created, the editable mode cannot be changed.

Meta information

A layout object can keep arbitrary meta data in the form of key/value pairs. This meta data is extracted during the reading of a layout and will reflect special properties of the layout file. For example, the GDS2 library name is available as meta information with key "libname".

The layout object offers methods to retrieve that information: Layout#each_meta_info will iterate over the meta data (returning a LayoutMetaInfo object). Layout#meta_info_value will get the value for a given name. Layout#add_meta_info will add a new meta information object and Layout#remove_meta_info will delete one.

Meta information is a different concept than properties.

Cell related methods

Cells can be created using the Layout#create_cell method. This method expects a cell name. If a cell with that name already exists, a new name is generated by appending a suffix. The method returns the Cell object of the new cell. Layout#rename_cell or Cell#name= can be used to change the name of a cell. The Cell object for a given name can be obtained with Layout#cell which returns the Cell object or nil if no cell with that name exists.

The cell name can be obtained from the cell index with the Layout#cell_name method or Cell#name). Layout#has_cell? can be used to determine whether a cell with the given name exists. Layout#is_valid_cell_index? can be used to determine whether a given index is a valid

cell index. Layout#cells returns the number of cells in the layout. To work with cells, the Cell object is required. It can be obtained from the cell index using the Layout#cell method.

The top cell of a layout can be obtained using Layout#top_cell. If multiple top cells exist, this method will raise an exception. In that case, Layout#top_cells can be used to obtain all top cells.

All cells in the layout can be iterated using the Layout#each_cell iterator. All top cells (cells which are not instantiated itself) can be iterated with the Layout#each_top_cell iterator. The cells can be iterated bottom-up (all child cells come before their parents) or top-down (all parents come before their children) using the Layout#each_cell_bottom_up or Layout#each_cell_top_down.

Layout#delete_cell deletes a cell. This method will keep the child cells, which may become top cells because their parent cell is deleted. Layout#delete_cells deletes multiple cells and is more efficient than deleting cell by cell. Layout#delete_cell_rec will delete a cell and all child cells (direct and indirect), irregardless whether they are used otherwise or not. Layout#prune_cell does the same but is somewhat more sensitive in that respect and does not delete child cells if they are instantiated by parents not in the cell tree below the cell being deleted. Layout#prune_subcells as prune_cell deletes the child cells or a cell but in contrast to prune_cell does not delete the cell itself.

Layer related methods

Layout#insert_layer creates a new layer in the layout. The layer will be available to all cells. This method receives a LayerInfo object which holds the information about the layer's name, layer and datatype. It returns a layer index which can be used to address shapes in the cells. Layout#insert_layer_at can be used to create a layer with a specific layer index, provided that index is not used yet.

Layout#is_valid_layer? can be used to determine whether a layer index is a valid index. Layout#layer_indices returns a list of indexes of all layers present. Layout#layers returns the number of layers present.

Layout#find_layerReturns the layer index for a given layer in various flavors. Layout#layer finds or creates a layer if it does not exist yet. Again, the layer can be given in various flavors - for example by layer and datatype, by name or with a LayerInfo object.

Layout#set_info can be used to change the LayerInfo object for a layer. Layout#get_info returns the LayerInfo object for a layer. To modify the information, obtain the information with get_info, modify it, and set the new information with set_info:

```
lv = RBA::Application::instance.main_window.current_view
ly = lv.current_cellview.layout
info = ly.get_info(0)
info.layer = 100
ly.set_info(0, info)
lv.add_missing_layers
lv.remove_unused_layers
```

The previous sample changes the layer number for layer index 0 to 100. "add_missing_layers" and "remove_unused_layers" will create new layer entries in the layer list and remove the entry for the previous layer.

For special purposes, special (temporary) layers can be created in the layout. Those layers basically behave like normal layers but don't appear in the layer list and are not saved to a file. Special layers can be created using Layout#insert_special_layer and Layout#insert_special_layer_at. Layout#is_special_layer? returns true if a given index is a special layer.

In addition, a layout contains a special layer which is used to implement the "guiding shape" feature of PCells. It is a special layer that serves as a container for shapes which parametrize PCells. The index of that layer can be obtained with Layout#guiding_shape_layer.

A list of the indexes for all layers inside the layout can be obtained with Layout#layer_indexes. A corresponding list of LayerInfo objects can be obtained with Layout#layer_infos.

Recursive full or region queries

A layout provides methods to retrieve shapes recursively. That means, that the shapes are delivered from all cells instantiated below a given top cell. Cells instantiated multiple times are also visited multiple times. While the shapes are delivered, information is provided what cell instances the specific shape instance is found in.

Recursive shape retrieval is done through a iterator, the RecursiveShapeIterator. This object delivers one shape each time. A RecursiveShapeIterator is created for example using the Layout#begin_shapes method. This method requires the cell index of the starting (initial) cell and a layer index. This code demonstrates how to use the RecursiveShapeIterator:

```
layout = RBA::Application::instance.main_window.current_view.active_cellview.layout
# start iterating shapes from cell "TOP", layer index 0
si = layout.begin_shapes(layout.cell_by_name("TOP"), 0)
while !si.at_end?
  puts si.shape.to_s + " with transformation " + si.trans.to_s
  si.next
```

```
end
```

The RecursiveShapeIterator's shape method delivers a shape reference (see description of the Shape class or Shape) which basically points to a shape inside a cell. Since the cell may be a child cell of the initial cell the RecursiveShapeIterator was created with, in general a transformation is present that tells how the cell's content shows up in the initial cell. The generic form of that transformation is a CplxTrans object which is delivered by the RecursiveShapeIterator's "trans" method. This transformation renders floating-point coordinates which is precise but not a suitable representation for transforming shapes into a form compatible with the database. For that purpose, a ICplxTrans object is provided as well through the "itrans" method. This transformation renders integer coordinates which may imply rounding effects in some cases. The RecursiveShapeIterator delivers the cell from which the current shape is taken through the "cell_index" method.

A RecursiveShapeIterator can be configured to retrieve only certain type of shapes (i.e. boxes, texts etc.). To do so, set the shape_flags attribute of the shape iterator before using it:

```
layout = RBA::Application::instance.main_window.current_view.active_cellview.layout
# start iterating shapes from cell "TOP", layer index 0
si = layout.begin_shapes(layout.cell_by_name("TOP"), 0)
si.shape_flags = RBA::Shapes::SBoxes
while !si.at_end?
  puts si.shape.to_s + " with transformation " + si.trans.to_s
  si.next
end
```

Also the maximum depth at which the RecursiveShapeIterator will traverse the hierarchy can be set using the "max_depth" attribute. Setting this attribute to 0 will report only shapes from the initial cell. The depth must be set before the first shape is retrieved.

The RecursiveShapeIterator can also deliver shapes from a region. The region is a rectangle specified in coordinates of the initial cell. To create a RecursiveShapeIterator that only delivers shapes inside that region use the Layout#begin_shapes_touching or Layout#begin_shapes_overlapping methods of the Layout object. These methods expect a Box object that specifies that rectangle. All shapes delivered will either touch or overlap that box when projected into the initial cell.

Shape manipulations should be avoided inside loops that iterate over shapes using a RecursiveShapeIterator. The reason is that shape manipulations may invalidate the internal state of the RecursiveShapeIterator. Instead, collect all shape references that need to be manipulated in an array and do the manipulations later.

Properties

As stated earlier, shapes can carry an arbitrary number of user properties in form of key/value pairs. For efficiency, these properties are not stored directly but in form of a property ID which identifies a unique set of properties. Retrieving a property hence requires an indirection over the property ID:

```
layout = RBA::Application::instance.main_window.current_view.active_cellview.layout
# first shape of cell "TOP", layer index 0
layer_index = 0
iter = layout.begin_shapes(layout.cell("TOP").cell_index, layer_index)
shape = iter.shape
# create a hash from the properties of that shape
props = Hash[*layout.properties(shape.prop_id).flatten]
# print the value of the property with key 1
puts props[1]
```

Since that scheme is somewhat tedious to use, a nice shortcut exists by using the "properties" method on the shape reference. This method implicitly modifies the property set and assigns a new property ID:

```
layout = RBA::Application::instance.main_window.current_view.active_cellview.layout
# first shape of cell "TOP", layer index 0
layer_index = 0
iter = layout.begin_shapes(layout.cell("TOP").cell_index, layer_index)
shape = iter.shape
# print the value of the property with key 1
puts shape.properties(1)
```

Changing a property requires to obtain a new property ID for the changed set:

```
layout = RBA::Application::instance.main_window.current_view.active_cellview.layout
# first shape of cell "TOP", layer index 0
layer_index = 0
iter = layout.begin_shapes(layout.cell("TOP").cell_index, layer_index)
shape = iter.shape
cell = layout.cell(iter.cell_index)
# create a hash from the properties of that shape
props = Hash[*layout.properties(shape.prop_id).flatten]
# change or add a property with key 1
props[1] = "NewValue"
# store the new properties
shape.prop_id = layout.properties_id(props.to_a)
```

For that problem also a shortcut exists. Use the "set_properties" method on the shape reference. This method implicitly modifies the property set and assigns a new property ID:

```
layout = RBA::Application::instance.main_window.current_view.active_cellview.layout
# first shape of cell "TOP", layer index 0
layer_index = 0
iter = layout.begin_shapes(layout.cell("TOP").cell_index, layer_index)
shape = iter.shape
# change or add a property with key 1 and value "NewValue"
shape.set_property(1, "NewValue")
```

A property ID of 0 in general indicates that no properties are attached. Please note that replacing a property ID and modifying the properties also invalidates any iterators and should not be done in a loop over shapes.

Cell instances and cells also carry a properties ID which can be used to assign user properties to cell instances. The cell instance properties ID is used like the shape properties ID. The shortcut methods "property", "set_property" and "delete_property" also are provided for cells and cell instances (Cell and Instance).

Note: The GDS format does not have string names for properties. As a result GDS only supports numeric property keys. OASIS, on the other hand, can handle string and numeric property "names". When saving layouts in GDS format, KLayout tries to convert the properties' names into numbers if possible (i.e. if it sees that the string is a number). If it can't, then the property is not saved. When the layout is read by KLayout upon opening, it gets converted to integer.

The LayerInfo class

The LayerInfo object encapsulates the layer's naming properties. In GDS, a layer is described by a layer number and datatype number. In OASIS, a text name can be added to that description. In other formats like DXF, a layer has just a text name.

The LayerInfo object thus has a twofold identity: a numeric identity (layer and datatype number) and a text layer name. Both properties can be specified. In that case, the numeric identity has precendence over the text name.

When a layout is loaded, LayerInfo objects are used to represent a layer's naming properties in the Layout object. The LayerInfo object associated with a layer can be retrieved using the Layout's "get_info" method. It can be set using the "set_info" method. In general, the LayerInfo property is detached from the layer index, so it can be assigned and manipulated freely.

The default constructor of LayerInfo will create a nameless object. Nameless layers are not saved to any file and can be used for internal purposes such as temporary or intermediate layers.

The LayerInfo#layer attribute allows read and write access to the layer number. LayerInfo#datatype is the attribute for the datatype number. LayerInfo#name gives access to the text name. LayerInfo#is_named? returns true, if the LayerInfo object represents a named layer (no layer or datatype number are specified). LayerInfo#is_equivalent? compares two LayerInfo objects and returns true, if both denote the same layer. This is not exact equivalence but follows the logical precendence: two layers are equivalent if layer or datatype number match (in that case the text name is ignored) or, if no layer and datatype number are specified, the name matches exactly.

LayerInfo objects supply a hash value (LayerInfo#hash) and can therefore be used as keys in Ruby hashes.

The Cell class

After the Layout object, the Cell object is the most fundamental object in KLayout's database API. It represents a cell, which itself is a collection of shapes per layer and instances of other cells. The methods provided by the Cell class deal with either the shape or the

instance aspect. A cell is also responsible for handling parts of the PCell scheme, either for the cell itself (if it is an incarnation of a PCell) or instances of PCells.

A cell has a name which can be retrieved using the Cell#name method and which can be set using the Cell#name= method. Setting the name is equivalent to using the Layout's "rename_cell" method. Cell#basic_name delivers the PCell or library cell name for cells imported from a library or PCell variants. "name" will deliver an internal unique name in that case. Cell#display_title is a string that encodes library name and PCell parameters as well and can be used as a descriptive title for the cell in user interfaces.

If the cell is a library cell (a "proxy"), it will have a database name (something like "TEXT$1") and a qualified name which states the library and the cell name separated with a dot (i.e. "Basic.TEXT"). The qualified name is not neccessarily unique. The qualified name of the cell can be obtained with Cell#qname.

A cell that represents a cell imported from a library or a PCell variant (or both) is called a proxy cell. For such cells, Cell#is_proxy? returns true. Such cells should not be manipulated since they may be refreshed when required and the original state is restored. Cell#is_library_cell? and Cell#is_pcell_variant? deliver a more detailed information about the nature of the proxy cell.

The layout that the cell lives in can be retrieved with the Cell#layout method. If the cell is created standalone (without layout), this method returns nil. In that case, a cell is not named either.

A cell can be a "ghost cell". A ghost cell is an empty cell which is not written to a layout write and created when a layout file is read with an unsatisfied reference. Unsatisfied references are present in some GDS files which represent partial layouts. By simple merging of two GDS files, such references can be made true instances, when another file contributes the cell for that reference. KLayout supports such unsatisfied references by providing the "ghost cells" which serve as a instance target but are not written. Ghost cells are simply cells where the Cell#is_ghost_cell? attribute is true. An empty cell can be made a ghost cell by setting the Cell#ghost_cell= property.

A cell has a bounding box which includes child cells as well. KLayout keeps a per-layer bounding box, so it's very simple to tell whether a cell is empty in a certain layer (including the hierarchy below). In that case the per-layer bounding box is an empty box. The overall bounding box can be derived with the Cell#bbox method. The per-layer bounding box can be derived with the Cell#bbox_per_layer method.

Cells can be marked as "ghost cells" using the Cell#ghost_cell=. Ghost cells are not saved into GDS files (but their references are). Also, ghost cells act as "placeholders" for cells - for example if a cell is pasted into a layout, it will replace any ghost cell with the same name. If a normal cell with the same name exists, a copy will be created instead. A cell can be asked whether it is a ghost cell using Cell#is_ghost_cell?.

Starting with version 0.23, cells can have properties as well, but writing cell properties to layout files is subject to some restrictions. Properties are only written to GDS if a special option is enabled because a potentially incompatible extension of GDS is used to store the properties. OASIS files support cell properties without restrictions.

Cells and shapes

A cell carries a set of geometrical shapes, organised in layers. A layer is specified by a layer index. The layer index is managed by the Layout object. It is basically an integer that identifies the layer in the layout. The shapes are stored in containers of the Shapes class. Given a layer index, the shapes object can be obtained from the cell through the Cell#shapes method:

```
shapes = cell.shapes(layer_index)
shapes.each do |shape|
  puts shape.to_s
end
```

The same can be achieved by directly iterating over the shapes in the cell:

```
cell.each_shape(layer_index) do |shape|
  puts shape.to_s
end
```

That iterator also allows specification of a filter so it delivers only a certain subset, i.e. only text objects. See Cell and Shapes for more details.

There are iterators that deliver shapes within a rectangular region, either overlapping or touching that region (Cell#each_overlapping_shape, Cell#each_touching_shape). They basically work like the "each_shape" iterator. Please note that these iterators are not recursive, i.e. they don't deliver shapes from child cells. Recursive iteration can be performed using Layout's "begin_shapes" method and the RecursiveShapeIterator object.

All shapes in a cell can be cleared using Cell#clear_shapes. A single layer can be cleared using the Cell#clear method with the layer index to clear. Both methods are not recursive, i.e. they only clear the shapes on the given cell, not on the child cells.

Layers can be copied using the Cell#copy method. The shapes of a layer can be moved to another layer using the Cell#move method. In both cases, the target layer is not overwritten, but the shapes from the source layer are added to the target layer. Cell#swap will swap the shapes of two layers. In all these cases, the operation is local on the cell, i.e. child cells are not affected.

Shapes within a cell are represented as Shape objects. A shape object is a generic object which represents either a polygon, a box, a path, a text or an edge object. Shapes provide some generic methods such as Shape#bbox to retrieve the bounding box or Shape#transform. Shapes act as "pointers" to geometrical objects inside the database. Manipulating a shape will also manipulate the database object.

To work with specific kind of shapes, the working classes such as Polygon are provided. Shapes can be converted into these working objects. The working objects are not related to the layout object and not having a connection to the layout makes them lightweight objects. A usual way of manipulating a shape is to translate it to a working object, modify that and assign the working object back to the shape.

Interfaces to floating-point working classes such as DPolygon are provided too. By convention such objects represent shapes in micrometer units. If such an object is requested from the shapes container it is converted from an integer-type object in database units to a floating-point type object in micrometer units by multiplying with the database unit. When sending such an object to the Shapes container the same happens in reverse. Internally, the integer type is used.

Cells and hierarchy

The direct children of a cell can be iterated with Cell#each_child_cell. That iterator delivers the cell index of each child cell of the cell. Similar, there is a Cell#each_parent_cell iterator. It delivers the cell indexes of all cells calling that cells. For a top cell that iterator delivers nothing. Cell#is_leaf? returns true, if the cell does not have child cells. Cell#is_top? returns true, if the cell is top cell.

Cells and their children form a directed acyclic cell graph. That means, no cell may instantiate a cell which itself calls the cell whether directly or indirectly. There is a set of cells called by a given cell, either directly or indirectly through children of the cell. That set is the "called" cell set. That cell set can be obtained with the Cell#called_cells method in form of an array of cell indexes.

Similar, there is a set of cells calling a cell, either directly or indirectly. That set of cells can be obtained with the Cell#caller_cells method. For a top cell that set is empty.

The number of hierarchy levels can be obtained with Cell#hierarchy_levels. This method delivers the length of the longest path to a leaf cell. A leaf cell has a hierarchy level count of 0.

A cell can be flattened using Cell#flatten. Child cells can be removed using Cell#prune_subcells. The cell can be removed with Cell#delete or Cell#prune_cell. The latter will also remove any child cells which are not used otherwise.

Copying information between cells

Instances can be copied from one cell to another using Cell#copy_instances. Shapes can be copied using Cell#copy_shapes. The latter method supports layer conversions by employing a LayerMapping object to specify input and output layers. In addition, shapes can be copied to another layout which automatically performs database unit conversion if necessary.

A full cell tree can be copied using Cell#copy_tree. This method will create a new hierarchy below the target cell matching the source cell's hierarchy and copy all shapes from source to target using that new hierarchy. Source and target cell may reside in different layouts and database unit conversion is done automatically.

The content of a cell can be copied to another cell hierarchically using Cell#copy_tree_shapes, provided a cell mapping exists. A cell mapping specifies, how child cells of the source cell are identified in the target, which can be a different layout. For the cell mapping a CellMapping object is employed. In addition, a LayerMapping object can be used to specify layer mapping to the target. If no cell mapping is provided, shapes are flattened into the next possible parent, which provides a way to creating flat copies of cells. "copy_tree" is a convenience method and is a special case of "copy_tree_shapes".

For all methods, "move" flavors are available (Cell#move_instances, Cell#move_shapes, Cell#move_tree and Cell#move_tree_shapes), which not only copy the information but also remove the respective objects in the source cell. That somewhat reduces the memory required for such operations.

Cells and instances

A cell also plays a role as container for the instances. An instance describes a cell that is placed into another cell. Technically an instance is a combination of a cell reference and a transformation. Raw instances are represented by CellInstArray objects. Instances inside a cell are referred to by Instance objects. An Instance object is basically a pointer into a CellInstArray stored inside a cell and associates properties with raw instances for example.

Instances can be created by using the various Cell#insert methods of Cell. Instances can have properties, so a property ID can be provided (see Layout class for a discussion about property ID's). Instances can be deleted using the Cell#erase method. All instances of a cell can be deleted using the Cell#clear_insts method.

An instance can be replaced by another CellInstArray object using the Cell#replace method. The properties ID can be changed using the Cell#replace_prop_id method. It is easier however to use the Instance object's "cell_inst=" or "prop_id=" method.

Instances can be iterated with Cell#each_inst. Cell#each_overlapping_inst and Cell#each_touching_inst delivers all instances overlapping or touching a given rectangular region. That means, the instance's overall bounding box overlaps or touches the search rectangle.

Cell#each_parent_inst delivers all parent instances. Parent instances are basically reverse instances represented by the ParentInstArray object. A parent instance identifies are parent cell and the instance of the given cell in that parent cell. This iterator allows deriving all instances of the given cell in other cells.

Instances can be transformed with a given transformation (either a orthogonal or a general complex transformation) using the Cell#transform method. This method invalidates the Instance pointer given as the argument and returns a new Instance pointer to the new cell instance.

A floating-point integer class exists too (DCellInstArray). By convention this object represent an instance in micrometer units. If such an object is requested from the cell it is converted from an integer-type object in database units to a floating-point type object in micrometer units by multiplying with the database unit. When sending such an object to the cell the same happens in reverse. Internally, the integer type is used.

Cells, libraries and PCells

If a cell is a cell imported from a library, Cell#is_library_cell? will return true and it is possible to derive the Library object from which this cell is imported using the Cell#library method. Cell#library_cell_index will return the cell index of the cell in the library's local layout.

If a cell is a PCell variant, either directly from the layout or from a library, Cell#is_pcell_variant? returns true. This method can also be called on an Instance object in which case it delivers true if the instance is an instance of a PCell. Cell#pcell_id returns the PCell declaration ID if the cell is a PCell variant. Cell#pcell_declaration will return the PCell declaration object. There is also a overload of "pcell_declaration" that determines the PCell declaration object for an Instance if it is a PCell instance. Cell#pcell_parameters delivers the PCell parameters for a cell (if it is a PCell variant) or an instance (if it is a PCell instance). The PCell parameters are an array of variable-type values and the interpretation is dependent on the PCell implementation. The PCellDeclaration object which can be obtained through "pcell_declaration" gives the necessary information about the interpretation of the parameters.

Finally, Cell#refresh allows refreshing the layout of a proxy cell, i.e. transfer the current state of a library cell into this cell or recompute the PCell layout. Usually this method needs not to be called. When PCell parameters change for example, the layout is automatically recomputed.

Cells and PCell instances

A cell can be a PCell variant as we've seen above. In addition, a cell can hold PCell instances. The parameters of PCell instances can be modified from the cell using Cell#change_pcell_parameter for individual parameters given by name or Cell#change_pcell_parameters for all parameters. For changing all parameters it is required to know the parameter's order and meaning. The order can be obtained from the PCell declaration class which itself can be retrieved from a PCell instance with Cell#pcell_declaration (with the instance as the first argument).

The PCell parameters of a PCell instance can be obtained with Cell#pcell_parameters (with the instance as the first argument). To get a specific parameter, use the PCell declaration object which lists the parameters on the order they are delivered by the "pcell_parameters" method.

The CellInstArray class

Despite its name, a CellInstArray object holds a cell reference which is not only an array, but also a single instances. The object represents a raw instance, in contrast to the Instance object which is basically a pointer to an instance inside the database. CellInstArray objects as raw instances can be created, copied, modified and stored in the usual containers, but once they are stored inside the Cell object, they can be addressed by the Instance object.

The CellInstArray object represents either single instances or array instances. Array instances correspond to GDS AREF records and are regular, two-dimensional (not necessarily orthogonal) arrays of instances. A single instance consist of a cell index, denoting the cell that is instantiated and a single transformation, which can be either a simple, orthogonal affine transformation without a magnification (a Trans object, see Trans) or a general affine transformation (a CplxTrans object, see CplxTrans). A cell instance array in addition specifies two dimensions (na, nb) and shift vectors (a, b). For each individual instance of the array, an additional displacement is added to the transformation which is computed by the following formula:

```
d=i*a+j*b   (i=0..na-1, j=0..nb-1)
```

A CellInstArray object that represents an array will return true on CellInstArray#is_regular_array?. In that case, the CellInstArray#a and CellInstArray#b attributes are the basic vectors of the array and CellInstArray#na and CellInstArray#nb are the dimensions of the array. CellInstArray#size is the number of instances in the array.

A CellInstArray with a simple transformation will return false on CellInstArray#is_complex? and the CellInstArray#trans attributes gives the basic transformation of the instance. If the transformation is complex, i.e. has a rotation angle which is not a multiple of 90 degree or a magnification, CellInstArray#is_complex? will return true and CellInstArray#cplx_trans should be used instead of CellInstArray#trans. In any case, CellInstArray#cplx_trans gives the correct transformation.

The CellInstArray#cell_index attribute gets or sets the cell index. The CellInstArray#bbox and CellInstArray#bbox_per_layer methods deliver the total bounding box of the instance including all instances for an array. CellInstArray#bbox_per_layer gives the bounding box for a single layer. Since the instance only knows the cell index, these methods require a Layout object in order to derive the actual cell's bounding box.

A CellInstArray object can be inverted using the CellInstArray#invert method. This method returns an array which represents the ways the parent cell is seen from the child cell. A CellInstArray object can also be transformed by a given transformation. The CellInstArray#transform method will (like CellInstArray#invert) transform the object in-place, i.e. modify the object. CellInstArray#transformed will do the same, but leave the object it is called on and return a modified copy (out-of-place). Various variants of the CellInstArray#transform and CellInstArray#transformed methods exist taking different forms of transformations (simple, complex).

The floating-point variant (DCellInstArray) behaves the same way, except that by convention the unit of coordinates is micrometers.

The Instance class

As stated earlier, the Instance object represents a cell instance in the database. Technically it acts as a proxy to some CellInstArray inside the database. In addition, it provides access to properties attached to the instance. Instance objects play an important role in the Cell class to identify a certain instance, for example to delete it.

The Instance class provides a couple of methods that give read access to the underlying CellInstArray object (Instance#a, Instance#b, Instance#na, Instance#nb, Instance#size, Instance#trans, Instance#cplx_trans, Instance#cell_index, Instance#cell, Instance#is_complex? or Instance#is_regular_array?). The whole CellInstArray object can be read with the Instance#cell_inst method. It is possible to copy (dup) and modify that object and replace the current CellInstArray with the new one using Instance#cell_inst=. Please note, that this operation may invalidate iterators and should not be done inside a loop using "Cell::each_inst" for example.

The cell the Instance object lives in can be obtained with Instance#cell. Instance#cell_index basically renders the same information, but in form of a cell index. The cell can be assigned (Instance#cell=) which changes is to refer to a different cell.

The layout the instance lives in can be obtained with Instance#layout. The cell the instance lives in is returned by Instance#parent_cell. The parent cell can be assigned (Instance#parent_cell=), which effectively moves the instance to a different cell.

User properties can be accessed through the Instance#prop_id attribute or, more convenient, through the Instance#property, Instance#delete_property or Instance#set_property methods. Please note that changing the property ID or the property values may invalidate iterators as well.

An instance has an equality operator. That operator returns true, if the Instances indentify the same object.

The Shapes class

The Shapes object is the basic container for geometrical shapes. It stores geometrical primitives (Boxes, Polygons, Paths, Texts and Edges) either directly or in compressed form to achieve a low memory usage. For example, OASIS shape arrays are stored as compact arrays when KLayout is used in viewer mode. The Shapes container provides a simplified view through the Shape object which is basically a pointer to an individual instance of a geometrical primitive. The Shapes container provides access to the primitives through Shape objects.

In editable mode (i.e. if a Shapes container lives in an editable Layout object), the shapes can be modified or deleted after they have been inserted. In the opposite mode (viewer mode), shapes can be added, but not modified nor deleted.

A Shapes container is usually obtained from a cell with a given layer index and is filled with geometrical primitives using one of the Shapes#insert methods. Please note that the shapes are specified in integer coordinates when you use the integer type objects and micrometer units when using the floating-point type objects (whose classes start with "D"):

```
# cell = a Cell object
# layer_index = the index of a the layer
shapes = cell.shapes(layer_index)
shapes.insert(RBA::Box::new(0, 0, 1000, 2000))
```

With floating-point objects:

```
# cell = a Cell object
# layer_index = the index of a the layer
shapes = cell.shapes(layer_index)
```

```
shapes.insert(RBA::DBox::new(0.0, 0.0, 1.0, 2.0))
```

A Shapes object can also be created without a cell:

```
shapes = RBA::Shapes::new
shapes.insert(RBA::Box::new(0, 0, 1000, 2000))
```

Standalone Shapes objects can be useful when the methods of the Shapes object are required (for example, region queries through "each_overlapping"). There are are variety of "insert" methods available, some of which copy shapes from other Shapes containers using a Shape object as the reference for the source object. Some of these variants allow one to specify a transformation which is applied before the shape is inserted. There are also variants for all geometrical primitives with or without a properties ID.

If a Shapes container is empty, Shapes#is_empty? will return true. The content of another Shapes container can be assigned to a Shapes object with the Shapes#assign method. The number of geometrical primitives inside the Shapes container can be obtained with the Shapes#size method. All shapes in the Shapes container can be deleted with the Shapes#clear method.

The content of the Shapes container can be iterated with the Shapes#each iterator. If will deliver Shape objects pointing to the current geometrical primitive. The Shapes#each_overlapping and Shapes#each_touching methods deliver only those primitives whose bounding box overlaps or touches the given rectangle. All iterators allow one to specify flags which confine the kind of shape delivered. The flags are a combination of the "S..." constants. For example:

```
# delivers all primitives:
shapes.each(RBA::Shapes::SAll) { |s| ... }
# delivers all primitives which have user properties attached:
shapes.each(RBA::Shapes::SAllWithProperties) { |s| ... }
# delivers only texts:
shapes.each(RBA::Shapes::STexts) { |s| ... }
# delivers only polygons and boxes:
shapes.each(RBA::Shapes::SBoxes | RBA::Shapes::SPolygons) { |s| ... }
```

A geometrical primitive inside the container can be erased using the Shapes#erase method. It is safe to erase shapes inside an iterator loop for editable containers.

Shapes can be replaced by other primitives using one of the methods. Please note that using "replace" inside an iterator loop may lead to unexpected behavior of the iterator, so modifying a shape inside an iterator loop should be avoided. Here is an example:

```
# DON'T:
# (replace polygons by their bounding boxes)
shapes.each do |shape|
  if shape.is_polygon?
    shapes.replace(shape, shape.bbox)
  end
end

# DO
# (replace polygons by their bounding boxes)
shapes_to_modify = []
shapes.each do |shape|
  if shape.is_polygon?
    shapes_to_modify.push(shape)
  end
end
shapes_to_modify.each do |shape|
  shapes.replace(shape, shape.bbox)
end
```

The latter solution requires some more memory but is in general safer. It is safe however to replace an object by the same kind of object inside a loop.

Shapes can be transformed by using one of the Shapes#transform methods provided by the Shapes object. Variants for simple and complex transformations exist. Please note that using a arbitrary-angle transformation on a box (i.e. a CplxTrans object with a rotation angle of 45 degree) will not render a rotated box since a box is by definition parallel to the axes. Instead this operation will render the bounding box of the rotated box. Transforming shapes is safe inside an iterator loop.

The Shapes container also manages user properties by employing properties ID's. Properties can be modified by obtaining a new ID from the layout object and replacing the property ID using the Shapes#replace_prop_id method. However it's much more convenient to use the Shape#property, Shape#set_property or Shape#delete_property methods of the Shape object. In both cases however, modifying the properties should be avoided inside an iterator loop.

All shapes inside a shape container can be transformed using Shapes#transform.

With version 0.23, new collection objects entered the stage: Region, Edges and EdgePairs which basically provide a way to store polygons, edges or edge pair objects independently from the shapes container. They are mainly used to implement bulk operations, specifically for the DRC functionality. The cooperate with the shapes container in the sense that they can be inserted into a shapes container which will produce polygons or edges in the layout database (edge pairs are converted to single edges or polygons. These classes are discussed in The Geometry API.

The Shape class

The Shape object provides a unified view to a geometrical primitive inside a Shapes container. It also plays an important role for addressing geometrical primitives inside a Shapes container. A Shape object has general methods and specific methods that apply depending on its identity.

General methods

The Shape#bbox method delivers the bounding box of the geometrical primitive addressed by the Shape object. Please note that the bounding box of a text only contains the single point of the text's origin, not the text drawing itself.

Shape#cell delivers the Cell object that shape lives in. The same way, Shape#layout delivers the Layout object and Shape#shapes the Shapes object.

Shape#property, Shape#set_property and Shape#delete_property allow modification of the user properties of the geometrical primitive. The same can be achieved somewhat less conveniently using a properties ID with the Shape#prop_id attributes' write accessor (Shape#prop_id=). Please note the comments in the description of the Shapes object regarding the interaction of these modifying operations with iterators. Shape#has_prop_id? returns true, if the shape has user properties attached.

Shape#type returns the type code of the shape addressed by the Shape object. This is a detailed code which distinguishes between different internal representations. It's more convenient to use one of the **is_...?** methods. For example Shape#is_box? returns true, if the geometrical primitive is a box. A note about Shape#is_polygon? and Shape#is_simple_polygon?: usually it is not required to distinguish between both. "is_polygon?" will also return true for simple polygons, so it is likely to be sufficient to just ask for "is_polygon?". Shape#is_user_object? returns true, if the primitive is a custom object. Such objects are rarely used and not supported by the Ruby API currently.

Shape#area delivers the area of the shape. The area is zero for a text object. Shape#perimeter delivers the perimeter of the shape. The perimeter is zero for a text object. The Shape object provides an equality operator which delivers true if two Shape objects point to the same primitive.

It is possible to replace a primitive by another one by using the shape assignment methods, i.e. Shape#box=, Shape#path=, Shape#polygon=, Shape#simple_polygon=, Shape#text= and Shape#edge=. Using these methods is equivalent to the using "replace" on the containing Shapes objects. Please see the notes on using "replace" inside iterators there.

Floating-point objects are supported too. For example, Shape#dbox is equivalent to Shape#box, but delivers the object in micrometer units. Shape#dbox= receives a micrometer-unit object. Shape#box_dp1 is equivalent to Shape#box_p1, but gets the first point of the box in micrometer units.

The layer index a shape is on can be obtained with Shape#layer. A shape can be moved to a different layer by assigning a different layer index with Shape#layer=. In that context, layers can also be addressed by layer/datatype or name using a LayerInfo object. The respective methods to address a shape's layer then are Shape#layer_info and Shape#layer_info=.

A shape can be transformed using one of the Shape#transform flavors.

Methods applying for box shapes

A Shape object represents a box if it returns true on Shape#is_box?. The only specific methods that are provided for box type shapes are the Shape#box getter and Shape#box= setter. Shape#box_center, Shape#box_center=, Shape#box_p1, Shape#box_p1=, Shape#box_p2 and Shape#box_p2= get or modify individual aspects of the box.

For the floating-point equivalents in micrometer units see the Shape.

Methods applying for polygon and simple polygon shapes

A Shape object represents a polygon or simple polygon if it returns true on Shape#is_polygon?. If the object is a simple polygon, it will also return true on Shape#is_simple_polygon?. A simple polygon is just a polygon that cannot have holes.

In every case, the Shape#each_edge iterator will deliver all edges (connections between points of the polygon). Shape#each_point_hull will deliver the points of the outer (hull) contour and Shape#each_point_hole will deliver the points of a specific hole contour. Shape#holes will deliver the number of holes. A simple polygon does not have holes and Shape#holes always gives zero.

The Shape#polygon getter delivers a Polygon object. The same way Shape#simple_polygon delivers a SimplePolygon object. A polygon with holes is converted in that case to a simple polygon by introducing cut lines to connect the holes with the outer contour.

The Shape#polygon= and Shape#simple_polygon= setters will replace the current object by the given new one.

For the floating-point equivalents in micrometer units see the Shape.

Methods applying for path shapes

A Shape object represents a path if it returns true on Shape#is_path?. The path's width can be obtained through the Shape#path_width method. The extensions of the path ends can be obtained with Shape#path_bgnext and Shape#path_endext for the start and end extensions. The Path object can be obtained with the Shape#path getter. For the path width and extensions setters are also provided (Shape#path_width=, Shape#path_bgnext= and Shape#path_endext=).

The path length can be obtained with Shape#path_length and includes the begin and end extensions. The round-ended path flag can be obtained with Shape#round_path? flag and set with Shape#round_path=.

The points of the path's spine can be iterated with Shape#each_point. Shape#polygon can be used to obtain the path's contour.

For the floating-point equivalents in micrometer units see the Shape.

Methods applying for text shapes

A Shape object represents a text object if it returns true on Shape#is_text?. The text's text string can be obtained with Shape#text_string. The text's origin and orientation is encoded in a transformation (a Trans object, see Trans) which can be obtained with the Shape#text_trans method. The font code, text size and alignment flags can be obtained with the Shape#text_font, Shape#text_size, Shape#text_halign and Shape#text_valign methods. See the description of the Text object for details about these attributes.

The text representation attributes can be set with Shape#text_font=, Shape#text_size=, Shape#text_halign= and Shape#text_valign=. Shape#text_trans= will modify the text's transformation.

Shape#text will deliver the Text geometrical primitive and Shape#text= allows replacing the shape with the given Text object.

For the floating-point equivalents in micrometer units see the Shape.

Methods applying for edge shapes

A Shape object represents an edge if it returns true on Shape#is_edge?. Edge objects in general are not well supported in KLayout currently. They can be created and manipulated by scripts, but cannot be drawn or modified on the user interface. In GDS files, edges are represented by zero-width paths which is sometimes breaking the conventions of other tools.

The only specific methods that are provided for edge type shapes are the Shape#edge getter and Shape#edge= setter.

For the floating-point equivalents in micrometer units see the Shape.

3.5. The Geometry API

The Geometry API

The central class of the layout database is the Layout class, which provides a concept for layers, shape containers, cells and hierarchy. Separate from that, a set of classes exist, which represent basic shapes and geometrical objects. While shapes embedded in a Layout object are not independent and care has to be taken when manipulating them inside the database API, the free objects are easy to manage, manipulate and in general to work with as they are normal objects in the script interpreter's context.

In addition, the gometry API provides basic objects such as points, edges and transformations for general use throughout the API. Higher objects such as regions and edge collections provide implementations for geometrical algorithms like boolean operations and DRC checks.

Most classes of the geometry API provide a hash value so they can be used as keys in Ruby hashes.

The Point class

The Point object represents an integer coordinate in the 2-dimensional layout space, expressed in database units. That object provides the x and y coordinates through the Point#x and Point#y attributes. Points can be added, subtracted, the euclidian distance and its square can be computed using the Point#distance and Point#sq_distance methods. The ***** operator used with a factor as the second operand will scale both the x and y coordinates.

Points and vectors

Next to points there is a corresponding vector class (Vector). A vector is basically the difference between two points. It is meant to describe the distance and direction between two points. The following rules therefore apply:

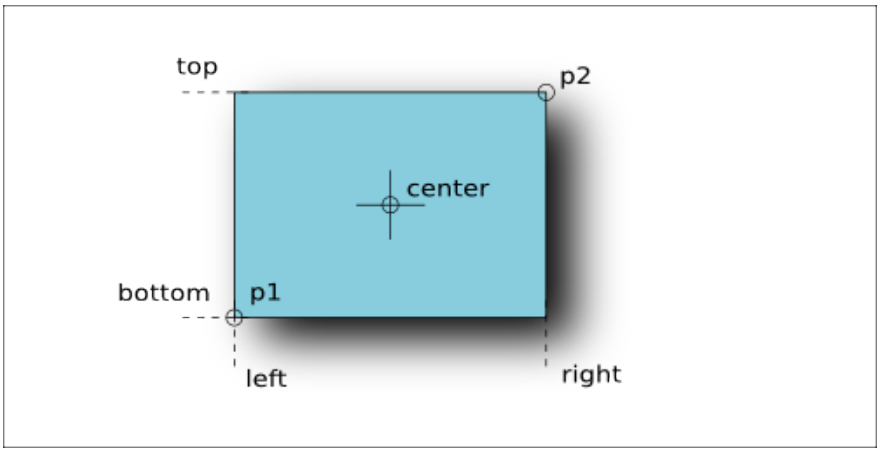
- Subtracting a point from a point renders a vector
- Adding a vector to a point renders a point

As points, vectors have an x and a y component which can be accessed with Vector#x and Vector#y. Vectors offer two functions to compute the vector product (Vector#vprod) and the scalar product (Vector#sprod). For some applications it's sufficient to know the sign of the product. You can get that with Vector#vprod_sign and Vector#sprod_sign respectively.

Vectors don't transform the same way than points. On transformation, only rotation, mirror and scaling (if applicable) is applied. Displacement is not applied. This way, the following two forms are equivalent:

```
(p1 - p2).transformed(t) == p1.transformed(t) - p2.transformed(t)
```

The Box class



The Box object represents a rectangle whose sides are parallel to the axes. The coordinates are integer values and express the rectangle's dimensions in database units. The basic specification consists of two points, giving the lower left and upper right corner. The constructor takes either two points (which are ordered internally) or four coordinates representing the left, bottom, right and top coordinates.

A box has a couple of attributes which are shown in the figure above. The Box#area method delivers the area of the rectangle. A box can be "empty" (non-existing). Such a rectangle can be created by the default constructor without parameters. An empty box basically behaves as a region without a point, i.e. testing the intersection with an empty box always returns false. An empty box returns true on Box#empty?. A box whose lower left point is identical to the upper right one contains just a single point. Such a box returns true on Box#is_point?.

A box supports a couple of operations:

- **Box & Box**: this operator computes the intersection of two boxes. If the boxes do not intersect, an empty box is returned.
- **Box * Box**: computes the "convolution" of two boxes. This operation can basically be viewed as the box which results when the first box is painted with a pen that has the size and displacement of the second box.
- **Box * factor**: scales the box, i.e. multiplies all coordinates with the given factor.
- **Box + Box**: computes the box that encloses both boxes given in the operation.
- **Box + Point**: computes the box that encloses the box and the point given as the second operand. It basically enlarges the first box so it includes the second point as well.
- **contains?(Point)**: returns true, if the point is inside the box or on its edges.
- **enlarge(Point)**: will enlarge the box by the x and y coordinates of the Point. Basically the x value of the point is subtracted from the left side and added to the right. The same way, the y coordinate is added to the top and subtracted from the bottom coordinate.
- **enlarged(Point)**: returns the enlarged box without modifying the box this method is called on (out-of-place operation).
- **move(Point)**: moves the box by the displacement given by the point. Basically the x value of the point is added to the left and right coordinate while the y coordinate is added to the top and bottom coordinates.
- **moved(Point)**: returns the moved box without modifying the box this method is called on (out-of-place operation).
- **inside?(Box)**: returns true, if the box the method is called on is inside the box given by the method's argument.
- **overlaps?(Box)**: returns true, if the given box overlaps with the box the method is called on.
- **touches?(Box)**: returns true, if the given box touches the box the method is called on.
- **transformed(Trans)**: returns the box transformed with the given transformation.
- **transformed(ICplxTrans)**: returns the box transformed with the given complex, integer-based transformation (see ICplxTrans). Note, that if the complex transformation includes a rotation by a non-90-degree angle (for example 45 degree), this operation does not return a rotated box, because by definition a box has edges which are parallel to the axes. Hence the general solution is to convert the box to a polygon:

```
# Wrong result
box = RBA::Box::new(0, 0, 100, 200)
transformed_box = box.transformed(RBA::ICplxTrans::new(1, 45, false, RBA::Vector::new))
# -> (-141,0;71,212)

# Correct result
transformed_box_as_polygon = RBA::Polygon::new(box).transformed(RBA::ICplxTrans::new(1, 45, false,
 RBA::Vector::new))
# -> (0,0;-141,141;-71,212;71,71)
```

- **transformed(CplxTrans)**: behaves like the previous "transformed" method but returns a floating-point coordinate object which is the target coordinate type of the CplxTrans object (see CplxTrans).

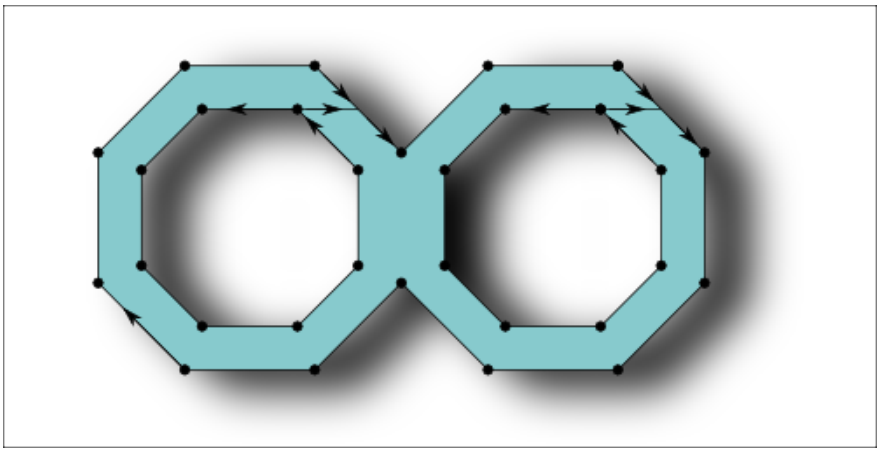
A box object can be constructed from a floating-point object (for floating-point objects see below). Lacking a database unit, no conversion from micrometer units to database units is performed. Instead, the floating-point coordinates are rounded to the nearest integer coordinates:

```
dbox = RBA::DBox::new(2.1, 3.1, 10.7, 11.8)
box = RBA::Box::new(dbox)
# -> (2,3;11,12)
```

An integer box can be turned into a floating-point unit box using Box#to_dtype

Floating-point boxes can be transformed using the DTrans, the DCplxTrans or the VCplxTrans transformations. The latter delivers an integer-type box and provides the reverse flavour transformation to "CplxTrans".

The SimplePolygon class



A SimplePolygon object is a polygon that does not have holes. It consists of a single, closed contour. Such polygons are compatible with the GDS2 format for example. A SimplePolygon object can be created from a Box object or from an array of Point objects. Internally, the points will be ordered into a clockwise orientation.

The class method SimplePolygon#from_dpoly creates a integer-coordinate type SimplePolygon object from a floating-point coordinate type DSimplePolygon object (see below). The floating-point coordinates are rounded to the nearest integer coordinates.

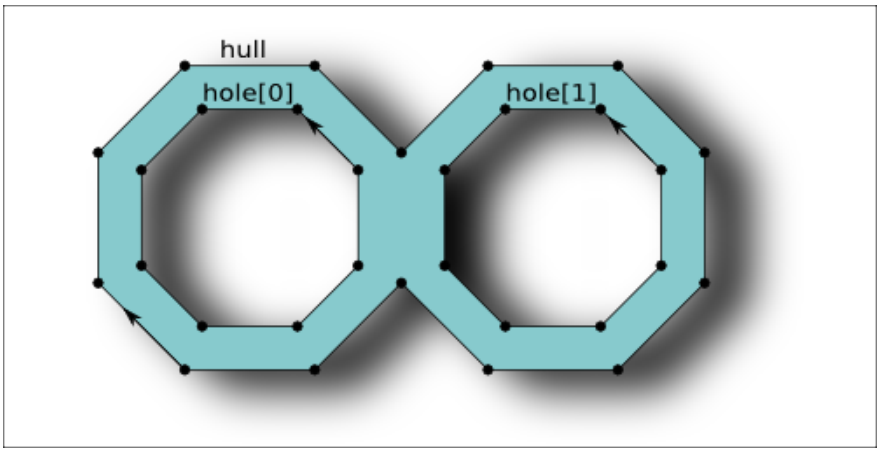
The SimplePolygon#bbox method returns the bounding box of the polyon. SimplePolygon#area will return the area of the polygon. SimplePolygon#num_points returns the number of points, while SimplePolygon#point returns the point for a given index. SimplePolygon#points= replaces the polygon by a new polygon with the given array of points.

The SimplePolygon#each_point iterator will deliver each point in clockwise orienation, starting form the bottom/leftmost one. SimplePolygon#each_edge will deliver all edges of the polygon (connecting every point with the next one).

SimplePolygon#compress will remove points that connect two collinear edges. It has a parameter that controls whether to remove reflecting edges (spikes) as well. SimplePolygon#inside? returns true, if a given point is inside the polygon. SimplePolygon#minkowski_sum computes the Minkowski sum between a polygon and another object in various flavors. SimplePolygon#move will displace the polygon by the distance given by the Point argument. SimplePolygon#moved will return the moved polygon without modifying the polygon it is called on (out-of-place operation). SimplePolygon#transformed will return the transformed polygon, either with a simple or a complex transformation (see the description of the Box object and the section about transformations below for a discussion of transformations). Finally, SimplePolygon#round_corners will apply a corner rounding to a copy of the polygon and return that copy without modifying the polygon.

Please note that using the Point-array constructors it is possible to create polygons with self-intersecting or twisted contours. Such polygons may not behave as expected.

The Polygon class



The Polygon object is basically an extension of the SimplePolygon object and in contrast to that object, supports holes. Polygon objects are not compatible with the GDS2 or OASIS format and will be converted to SimplePolygon objects by introducing cutlines when writing them.

A polygon consists of an outer contour (the hull) and zero to many hole contours. The outer contour like for the SimplePolygon is oriented clockwise while the hole contours are oriented counterclockwise. This orientation is established internally. For the API, contours are represented by arrays of Point objects.

A Polygon differs from a SimplyPolygon by providing methods to modify holes and the hull. Holes can be inserted by using the Polygon#insert_hole method. A hole is identified by an index. The Polygon#holes method returns the number of holes. The hole index runs from 0 to the number of holes minus one.

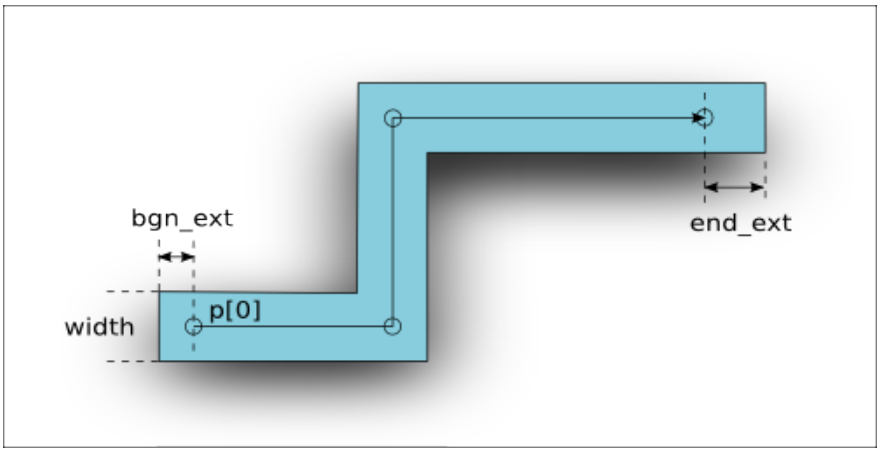
The class method Polygon#from_dpoly creates a integer-coordinate type Polygon object from a floating-point coordinate type DPolygon object (see below). The floating-point coordinates are rounded to the nearest integer coordinates. A constructor is provided that creates a Polygon object from a SimplePolygon object.

Polygon#each_edge will deliver all edges. The orientation of the edges determines whether they belong to a hole or the hull contour. Polygon#each_point_hull will iterated over all points of the hull and Polygon#each_point_hole over the points of the hole with the given index. Polygon#num_points_hull will return the number of points for the hull and Polygon#num_points_hole the number of points for the given hole. Polygon#point_hull will return a specific point from the hull and Polygon#point_hole a specific point for the given hole. Polygon#num_points returns the total number of points.

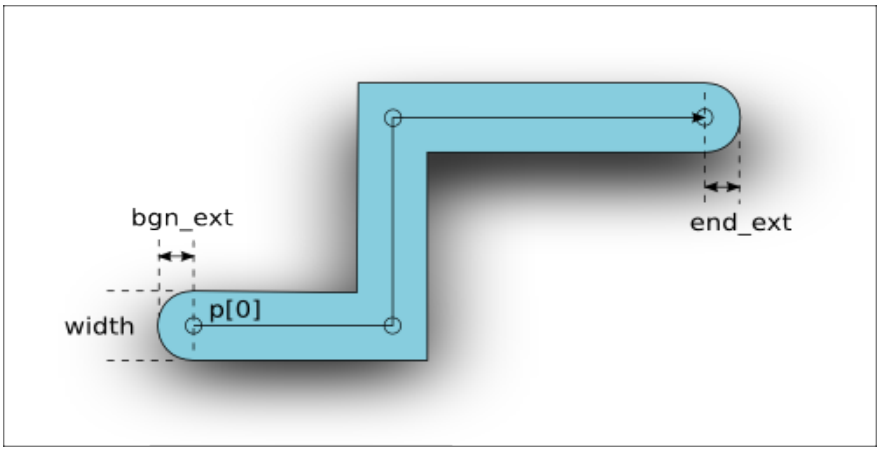
Polygon#resolve_holes will remove all holes and connect them with the hull by introducing cutlines that connect the hole with the hull. This operation introduces new vertexes and hence may apply some distortion due to grid snapping. Polygon#resolved_holes returns the polygon with the holes removed without modifying the object it is called on (out-of-place operation). Polygon#to_simple_polygon basically does the same but returns a SimplePolygon object.

Using Polygon#assign_hull (or Polygon#hull=) and Polygon#assign_hole the hull contour or a hole contour can be replaced with the given array of Point objects. Please note that it is possible to create invalid polygons where the holes are not completely contained in the hull. Such polygons may not behave as expected. The same is true for polygons with self-intersecting or twisted contours.

The Path class



A Path object represents a line with a certain width. The figure above depicts the basic properties of a path. The basic geometry of a path is defined by its spine - a sequence of points that the path follows. By default, a path has rectangular end caps with variable length. The end caps can be round, in which case the extension should be half the path width to avoid paths which are not compatible with the GDS2 format. Round-ended paths return true on Path#is_round?. An example for such a path is depicted in the following figure:



A path with is given in database units. The hull contour of paths with an odd width cannot be represented on-grid and should be avoided. Such path objects are allowed in the GDS2 format, but not in OASIS.

The spine points of a path can be iterated with Path#each_point. Path#num_points returns the number of points. Path#points= allows replacing of the spine with the given array of Point objects. Path#round= sets the "round ended" flag. Path#bbox and Path#area deliver the bounding box and the area, where the area is only approximate and is computed from the spine's length including the extensions times the width for efficiency. For certain acute-angle configurations that value may not be the exact area.

Path#polygon returns the polygon representing the path's hull. Path#simple_polygon returns a SimplePolygon object that represents the hull. Path#move, Path#moved and Path#transformed basically work like for the other objects.

The class method Path#from_dpath creates a integer-coordinate type Path object from a floating-point coordinate type DPath object (see below). The floating-point coordinates are rounded to the nearest integer coordinates.

The Text class

A Text object is basically a point with a label attached. The extension of a text object only includes the point, not the text itself. For display purposes, a text orientation, font and alignment options can be specified.

The location and the orientation of the text are combined in a Trans object. That transformation can be read and written with the Text#trans attribute. The text orientation is not always shown in the drawing. Whether the text is shown in the orientation specified depends on the application settings.

Font number and alignment flags can be read and written using the Text#font, Text#halign and Text#valign attributes. If one of these attributes is not set, the application-provided default is used. The font is currently just a number and is provided to support the respective property of the GDS2 format. For the documentation of the alignment flags see the class documentation for the Text class.

The text string is accessible with the Text#string attribute. Not all characters are supported. Depending on the format, only a subset of the ASCII character set are supported. Sometimes, line-feed control characters are found in text strings. Following the strict OASIS specification for example, such characters are not allowed. The lower and upper case letters and most of the special printing characters of the ASCII character set are usually safe for use in text strings.

Text#move, Text#moved and Text#transformed basically work like for the other objects.

Passing a DText object to the Text constructor creates an integer-coordinate type Text object from a floating-point coordinate type DText object (see below). The floating-point coordinates are rounded to the nearest integer coordinates.

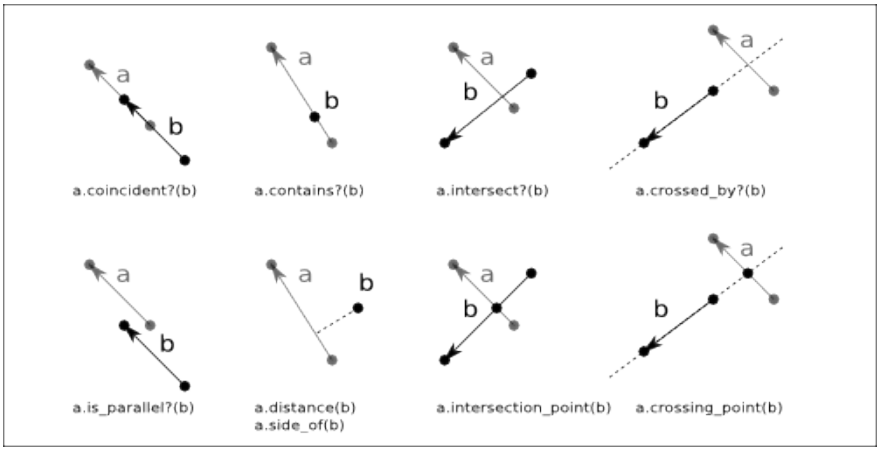
The Edge class

Edge objects are basically connections between two points. Edge objects are provided to support special applications and are mapped to zero-width, two-point paths in the GDS2 format. Edge objects are not supported as editable objects in KLayout currently. Edge objects may be created by script and are useful sometimes to represent the output of a design rule check tool for example.

Edge objects however a frequently used as raw objects in various applications, i.e. as one output option of a boolean operation. Therefore, the Edge class provides a couple of operations on edges, mainly tests for the geometrical relationship of two edges and points in relation to edges. An edge also may represent a straight line through the two endpoints of the edge, i.e. for the Edge#crossed_by? check.

An edge is defined by two points: the start and end point. Edge#p1 is the attribute for the start point, Edge#p2 the attribute for the end point. Various properties deliver extensions of the edge (Edge#dx and Edge#dy for the horizontal and vertical extension, Edge#length and Edge#sq_length for the length and the square of the length, Edge#ortho_length for the Manhattan distance between the points). Edge#swap_points swaps p1 and p2 and effectively inverts the orientation of the edge. Edge#bbox is the bounding box of the edge.

Various methods test the relationship between two edges or an edge and a point. See the following figure for a summary of these methods:



Edge#move, Edge#moved and Edge#transformed basically work like for the other objects. The class method Edge#from_dedge creates a integer-coordinate type Edge object from a floating-point coordinate type DEdge object (see below). The floating-point coordinates are rounded to the nearest integer coordinates.

The floating-point geometrical primitives

Beside the integer-coordinate primitives like Box, Edge, Polygon etc., KLayout provides floating-point coordinate variants as well. These objects require more memory and are subject to floating-point rounding issues and are therefore not employed inside the database. They are provided however, to allow a temporary representation of micron-unit objects or for use int the Marker class for example.

The class names for the floating-point variants is the same the integer-coordinate type prefixed with a "D" (for example, DBox is the floating-point variant of Box). Since floating-point variants support fractional coordinates, scaling with an arbitrary value is not connected

with a loss of accuracy due to rounding. That is why floating-point coordinates are the target type of general transformations including an arbitrary scaling. For example:

```
RBA::Box.new(1,1,2,2)*2.5
# -> (3,3;5,5)
# but:
RBA::DBox.new(1,1,2,2)*2.5
# -> (2.5,2.5;5,5)
```

However, the higher precision of floating-point coordinates comes with subtle issues originating from the finite-precision representation. For example, the value of 0.1 cannot be precisely represented by a floating-point value. The value of 0.1 is approximated and therefore depending on the way it is approximated, 0.1 is not necessarily equal to 0.1. For example (try this in the Ruby console):

```
a = 0.1
b = 1e-6*1e5
a
# -> 0.1
b
# -> 0.1
# but:
a == b
# -> false!
# the reason is:
"%.21g"%a
# -> 0.100000000000000005551
"%.21g"%b
# -> 0.099999999999999916733
```

Because of that, direct comparison of coordinate values should be avoided. The internal precision used is 0.00001. By convention, floating-point type objects are supposed to be used for micrometer-unit objects, hence this precision corresponds to 0.01 nm which corresponds to the tenth of an atom and should be well below physical tolerances.

Some operations which are implemented on integer coordinates (like the removal of holes in polygons) are not available for the floating-point type objects.

All floating-point type objects have a "to_itype" method (i.e. DBox#to_itype) which convert the floating-point type object to the integer-coordinate type object. This method can be given a database unit for scaling from micrometer units. Integer-coordinate type objects can be converted to floating-point type objects the same way through the "to_dtype" method (i.e. Box#to_dtype. A database unit can be passed to this method too for conversion to micrometer units. These methods however require some rounding and are therefore responsible for a potential distortion of the geometry of the object.

For the point and vector object there is also a floating-point equivalent (DPoint and DVector). Both behaves like their integer-coordinate equivalent.

Transformations

Transformations in KLayout are restricted affine transformations. Shear transformations and anisotropic scaling is not supported. All transformation in KLayout follows the conventions implied by the GDS2 and OASIS formats and include in that order:

- Mirroring around the x axis (optional)
- Rotation
- Scaling
- Displacement

For memory performance, a restricted version of that transformation is used if possible. In that restricted version, the rotation angles are confined to a multiple of 90 degree and scaling is not supported. This restricted affine transformation is provided through the Trans class. The 8 possible rotation/mirror variants can be coded in a single rotation/mirror code which is used frequently throughout KLayout (see Transformations in KLayout).

For transforming floating-point coordinates, the DTrans object is provided. The basic difference is that the displacement uses floating-point coordinates (by employing a DVector object).

To support more complex affine transformations include arbitrary-angle rotations and scaling, the complex transformation objects are provided. In addition, the complex transformation object can translate between integer and floating-point coordinate types.

- CplxTrans: takes integer coordinates and delivers floating-point coordinates. Therefore it uses a DVector for the displacement. The inverse of this transformation is a VCplxTrans class (see below).
- DCplxTrans: takes floating-point coordinates and delivers floating-point coordinates. Therefore it also uses a DVector for the displacement.
- ICplxTrans: takes integer coordinates and delivers integer coordinates. Therefore it uses a Vector for the displacement. This transformation is convenient to provide complex transformations for database operations but implies rounding errors due to rounding to integer coordinates on output. It is safe to use however for integer-factor scaling operations for example.
- VCplxTrans: takes floating-point coordinates and integer coordinates. Therefore it also uses a Vector for the displacement. The inverse of this transformation is a CplxTrans object.

Multiplication of a transformation with an object renders the transformed object:

```
t = ... # a transformation
p = ... # some point

# compute the transformed point:
q = t * p
```

Multiplication of two transformations corresponds to concatenation of two transformations:

```
t1 = ... # a transformation
t2 = ... # another transformation

# Multiplication of t2 and t1 renders an equivalent transformation
# that corresponds to "apply t1 first and then t2":
(t2 * t1) * p == t2 * (t1 * p)
```

When multiplying two complex transformations, the resulting transformation type will have the corresponding input and output types. For example when multiplying a VCplxTrans with a CplxTrans, a ICplxTrans object will be created. This is because the first transformation takes integers and the second one delivers integers. Hence the resulting transformation is ICplxTrans.

The Region class

Regions are basically collections of polygons. Regions provide higher functions such as boolean operations and sizing which cannot be implemented on pure polygons because their output may be a number of polygons. A Region is a general representation of a set of points in a two-dimensional area while a polygon is a coherence set of points.

Regions can be created by starting with an empty region and filling the latter with polygons. Regions can also be created from a RecursiveShapeIterator which allows feeding layout data into the region in a very flexible way. In the latter case, boxes and paths will be converted to polygons when feeding them into the region.

Regions feature some important concepts:

- **Merged semantics:** A region can be created from a series of polygons which potentially overlap or touch. In "merged semantics" the region will merge the polygons forming big polygons from touching ones and removing overlaps.
- **Minimum coherence:** In certain cases, the output of merging polygons is ambiguous. In the "kissing corner" case, the touching polygons may be either considered separate ("minimum coherence") or belonging together.
- **strict mode:** In strict mode, some operations are performed even if they are not necessary. For example, an XOR between a region and an empty region will render the first input. Hence the implementation can simply copy the first input in that case. With strict mode, the operation will be performed in every case which is less efficient but renders merged polygons.

The region object is very mighty and easy to use. Here is an example which computes the difference of two boxes (rendering a ring) and sizes the latter:

```
r1 = RBA::Region::new
r1.insert(RBA::Box::new(-2000, -3000, 2000, 3000))

r2 = RBA::Region::new
r2.insert(RBA::Box::new(-1500, -2000, 1500, 2000))

r = (r1 - r2).sized(100)

puts r
```

Using a Region object with input from a cell is almost as simple as this. The following example will take the input from layer 11 and 21 from the top cell and the hierarchy below (as is flat) and compute the intersection in layer 100:

```
ly = RBA::CellView::active.layout

l11 = ly.layer(11, 0)
l21 = ly.layer(21, 0)

r11 = RBA::Region::new(ly.top_cell.begin_shapes_rec(l11))
r21 = RBA::Region::new(ly.top_cell.begin_shapes_rec(l21))

ly.top_cell.shapes(ly.layer(100, 0)).insert(r11 & r21)
```

A variety of operations is implemented on regions:

Region#&, Region#|, Region#- and Region#^ implement boolean operations (AND, OR, NOT and XOR). The operations can be combined with assignment (in-place) using Region#&= etc. Region#+ adds the polygons from another region to self which is basically the same then a boolean OR.

Region#area and Region#bbox deliver the area and bounding box of the region.

Region#each will deliver all original polygons (unless the region was already merged).

Region#each_merged will deliver the merged polygons.

Region#edges will translate the polygons to edges covering their boundaries. If merged semantics is specified, the edges will only cover the outer edges, not inner ones.

Region#enclosing_check, Region#inside_check, Region#isolated_check, Region#notch_check, Region#separation_check, Region#space_check and Region#width_check implement DRC functions. DRC functions deliver EdgePairs edge pair collections, not regions. Each edge pair marks a violation of the given check.

Region#extents replaces each polygon with its bounding box.

Region#grid_check performs an on-grid check returning edge pairs for off-grid markers.

Region#holes identifies holes and delivers a new region with the holes as filled polygons.

Region#hulls identifies holes and delivers a new region without the holes.

Region#insert adds polygons in various flavors.

Region#interacting and Region#not_interacting select polygons interacting (overlapping or touching) or not interacting with polygons from another region.

Region#members_of selects polygons which are contained in the same way in another region.

Region#merge and Region#merged merge the polygons. This feature allows selecting overlapping polygons (minimum wrap count parameter).

Region#minkowski_sum computes the Minkowski sum of the region with other objects (edges, single polygons and other regions).

Region#move, Region#moved, Region#transform and Region#transformed apply geometrical transformations.

Region#rectangles, Region#rectilinear, Region#non_rectangles and Region#non_rectilinear filter out polygons by their appearance.

Region#inside, Region#outside, Region#not_inside and Region#not_outside filter out polygons by their relation to the polygons in the other region.

Region#round_corners and Region#rounded_corners apply corner rounding to polygons.

Region#size and Region#sized will size the polygons (shift edges).

Region#smooth smoothes out coarse steps of the polygons.

Region#snap and Region#snapped apply grid snapping.

Region#with_angle marks polygon vertices which satisfy a certain angle criterion.

Region#with_perimeter, Region#with_area and many more "with_..." methods select polygons based on their properties.

The Edges class

Edges represents a collection of edges which comprise either full polygons (forming closed contours) or parts of polygons. Edges can be derived from Region objects using the Region#edges

A variety of operations is implemented on regions:

Edges#&, Edges#|, Edges#- and Edges#^ implement boolean operations (AND, OR, NOT and XOR). The operations can be combined with assignment (in-place) using Region#&= etc. Edges#+ adds the polygons from another edge set to self which is basically the same then a boolean OR. Boolean AND and NOT are available between edges and regions as well and will deliver the edge parts inside the given region or not inside the given region.

Edges#length and Edges#bbox deliver the total length and bounding box of the edge set.

Edges#each will deliver the edges in the collection.

Edges#centers will deliver the center parts of the edges.

Edges#enclosing_check, Edges#inside_check, Edges#separation_check, Edges#space_check and Edges#width_check implement DRC functions. DRC functions deliver EdgePairs edge pair collections, not edge sets. Each edge pair marks a violation of the given check.

Edges#start_segments and Edges#end_segments replaces each edge with a part at the beginning or end.

Edges#extended, Edges#extended_in and Edges#extended_out create polygons on that represent the edge with a certain width.

Edges#extents returns a region with the bounding boxes of the edges.

Edges#inside_part and Edges#outside_part return the parts of the edge set which are inside or outside a given region. The effect is comparable to the boolean operations but differs at the exact boundary of the polygons of the region.

Edges#interacting and Edges#not_interacting select edges interacting (overlapping or touching) or not interacting with edges of another edge set or polygons from another region.

Edges#members_of selects edges which are contained in the same way in another edge set.

Edges#merge and Edges#merged merge (join) the edges.

Edges#move, Edges#moved, Region#transform and Edges#transformed apply geometrical transformations.

Edges#with_length, Region#with_angle and many more "with_..." methods select edges based on their properties.

The EdgePair class

Edge pairs are handy objects to describe the output of a DRC violation. A space violation marker for example will consist of two edges which mark the opposite parts of the space violation. Such markers are represented by objects of EdgePair.

Edge pairs are simple objects consisting of two edges ("first" and "second"). Polygons can generated from edge pairs covering the marked edges and their connecting files using EdgePair#polygon. Edge pairs can be transformed using EdgePair#transformed.

Edge pairs can be "normalized" using EdgePair#normalized. This method returns the normalized version of the edge pair. Normalization will bring the edge pairs in a form such that when connecting their start and end points a closed loop without intersections is formed. Normalized edge pairs will produce nicer polygons later on.

A floating-point version of the EdgePair class exists as well: DEdgePair.

The EdgePairs class

EdgePairs provides a collection of edge pairs and is the preferred output of DRC functions which deliver one DRC marker for each violation.

EdgePair collections can be conveniently split into individual edges using EdgePairs#edges or converted into polygons covering the markers with EdgePairs#polygons. The first and second edges can be extracted as a Edges edge collection with EdgePairs#first_edges and EdgePairs#second_edges. EdgePairs#extents will deliver a region containing the bounding boxes of the edge pairs.

3.6. Events And Callbacks

Introduction

In some places, the API requires to attach code to an event. An event could be a menu item which is selected or a change of some status which might require some action. The API allows implementation of specific code which is called in that case. This enables us to implement the functionality behind a menu item. In this text we will refer to such functionality by the general term "callback". In general a callback is custom code that is called from the API in contrast to API code that is called from the custom code.

There are basically two ways to attach specific code to a callback:

- **Reimplementation**: some API classes provide "virtual" methods. "virtual" is a C++ term and means a method that can be overridden in a derived class. This technique is employed for example in the "Strategy" design pattern. In strictly typed C++ this is quite a common pattern which allows definition of interfaces and concrete implementations based on those interfaces. Ruby as a dynamic language doesn't care much about classes and their relationship: an object either has a method or it hasn't. So, reimplementation is just a matter of providing the right method. An examples for the strategy pattern is the BrowserSource class (BrowserSource).

- **Events**: events allow attaching a piece of code to an event. In Ruby, such a block is "Proc" object, in "Python" it is a "callable" object ("lambda function" is a term used in both languages for this kind of concept). In case the event is triggered, this attached code is executed. Multiple lambda functions can be attached to the same event and removed from the latter. Events can be cleared of attached code, where only the blocks attached from one language can be cleared together - code attached from Python cannot be cleared from Ruby. An example for events is the Action class (Action) which provides both the reimplementation interface ("triggered" method) and the event interface ("on_triggered"). By the way, Qt signals are mapped to events in KLayout's Qt binding (The Qt Binding).

The "Observer" class which was there prior to KLayout 0.25 has been dropped in favour of the more flexible events. It is no longer supported.

Reimplementation (Strategy Pattern)

The BrowserSource (BrowserSource) class is a nice example for the Strategy pattern. It is used by the BrowserDialog class (BrowserDialog) as a kind of internal HTML server which handles URL's starting with "int:". For this, a script has to provide a class that reimplements the "get(url)" method. In the following example, a BrowserSource is created that takes an URL with an integer index number and delivers a HTML text with a link to the URL with the next index.

Here is the code. This example demonstrates how the "get" method is reimplemented to deliver the actual text.

```
module MyMacro

  include RBA

  class MyBrowserSource < BrowserSource
    def get(url)
      next_url = url.sub(/\d+/) { |num| (num.to_i+1).to_s }
      "This is #{url}. <a href='#{next_url}'>Goto next (#{next_url})</a>"
    end
  end

  dialog = BrowserDialog::new
  dialog.source = MyBrowserSource::new
  dialog.home = "int:0"
  dialog.exec

end
```

```
from pya import BrowserSource, BrowserDialog

class MyBrowserSource(BrowserSource):
  def get(self, url):
```

```
    next_url = "int:" + str(int(url.split(":")[1]) + 1)
    return f"This is {url}. <a href='{next_url}'>Goto next ({next_url})</a>>"

dialog = BrowserDialog()
dialog.home = "int:0"
dialog.source = MyBrowserSource()
dialog.exec_()
```

Ruby even allows reimplementation of a method without deriving a new class, because it allows defining methods per instance:

```
module MyMacro

  include RBA

  source = BrowserSource::new
  def source.get(url)
    next_url = url.sub(/\d+/) { |num| (num.to_i+1).to_s }
    "This is #{url}. <a href='#{next_url}'>Goto next (#{next_url})</a>"
  end

  dialog = BrowserDialog::new
  dialog.source = source
  dialog.home = "int:0"
  dialog.exec

end
```

Events

Events are the callback variant which is the easiest one to use. Using an event it is possible to directly attach a block of code to a callback. An event has a specific signature, i.e. the parameters it provides. The block can obtain this parameters by listing them in its argument list.

Here is a simple example that uses the parameterless "on_triggered" event of the Action class (Action). It puts a new entry into the tool bar and if it is clicked, it displays a message box:

```
module MyMacro

  include RBA

  action = Action::new
  action.on_triggered do
    MessageBox::info("A message", "The action was triggered", MessageBox::Ok)
  end
  action.title = "My Action"

  Application::instance.main_window.menu.insert_item("@toolbar.end", "my_action", action)

end
```

The Python version is:

```
from pya import Action, MessageBox, Application

def on_triggered():
  MessageBox.info("A message", "The action was triggered", MessageBox.Ok)

action = Action()
action.on_triggered = on_triggered
action.title = "My Action"
```

```
Application.instance().main_window().menu().insert_item("@toolbar.end", "my_action", action)
```

Specifying a block to an event will make the event only execute that block. A more flexible way of controlling the code attached to events is available through the += and -= operators:

```
module MyMacro

  include RBA

  code = lambda do
    MessageBox::info("A message", "The action was triggered", MessageBox::Ok)
  end

  action = Action::new
  action.on_triggered += code

  ...

  # to remove the code from the event, use:
  action.on_triggered -= code

  # to replace all event handlers by the one given by "code":
  action.on_triggered = code

  # to clear all event handlers use:
  action.on_triggered.clear
```

Synonyms for the += operator are `add` and `connect`. The latter makes code more familiar for PyQt users. In the same way, synonyms for the `-=` operator are `remove` and `disconnect`.

If the Qt binding is available (see The Qt Binding), Qt signals are implemented as events. This way it's very simple to create a Qt dialog. In following example, the "textChanged" signal of QLineEdit is attached a code block which copies the text of the input field to the label below:

```
module MyMacro

  include RBA

  dialog = QDialog::new(Application::instance.main_window)
  layout = QVBoxLayout::new(dialog)
  input = QLineEdit::new(dialog)
  label = QLabel::new(dialog)
  layout.addWidget(input)
  layout.addWidget(label)

  # implement the textChanged signal as event:
  input.textChanged { |text| label.text = text }

  dialog.exec

end
```

The Python version is:

```
from pya import QDialog, QVBoxLayout, QLineEdit, QLabel, Application

dialog = QDialog(Application.instance().main_window())
layout = QVBoxLayout(dialog)
input = QLineEdit(dialog)
label = QLabel(dialog)
layout.addWidget(input)
layout.addWidget(label)
```

```
def text_changed(text):
  label.text = text

# implement the textChanged signal as event:
input.textChanged = text_changed

dialog.exec_()
```

Using the += operator on the event, multiple handlers can be added to a signal:

```
module MyMacro

  include RBA

  dialog = QDialog::new(Application::instance.main_window)
  layout = QVBoxLayout::new(dialog)
  input = QLineEdit::new(dialog)
  label1 = QLabel::new(dialog)
  label2 = QLabel::new(dialog)
  layout.addWidget(input)
  layout.addWidget(label1)
  layout.addWidget(label2)

  # two signal consumers:
  input.textChanged += lambda { |text| label1.text = text }
  input.textChanged += lambda { |text| label2.text = text.reverse }

  dialog.exec

end
```

with the Python version:

```
from pya import QDialog, QVBoxLayout, QLineEdit, QLabel, Application

dialog = QDialog(Application.instance().main_window())
layout = QVBoxLayout(dialog)
input = QLineEdit(dialog)
label1 = QLabel(dialog)
label2 = QLabel(dialog)
layout.addWidget(input)
layout.addWidget(label1)
layout.addWidget(label2)

def text_changed1(text):
  label1.text = text

def text_changed2(text):
  label2.text = text[::-1]

# two signal consumers:
input.textChanged += text_changed1
input.textChanged += text_changed2

dialog.exec_()
```

3.7. The Ruby Language Binding

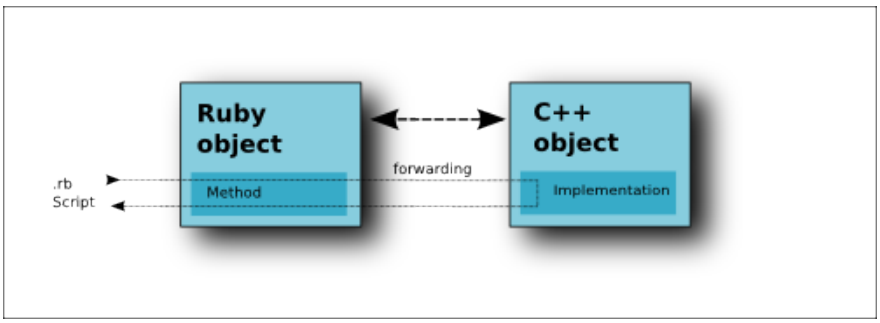
This article covers the basics of the Ruby binding provided by KLayout. The Ruby binding is basically a way to access the native code classes of KLayout through a Ruby interface. KLayout is written in C++, hence the topic covered here is the interface between C++ and Ruby objects. The Ruby API (RBA) is based on the Ruby binding of a selection of C++ classes. RBA is also the framework that implements the Ruby side of the binding. The C++ side is a more generic form which is not strictly confined to a certain programming language. The C++ side of the framework is referred to as GSI (generic scripting interface) in the KLayout sources.

Joining Two Worlds: Ruby and C++

The usual and most simple case of a Ruby/C++ binding is a Ruby wrapper over a C++ object. When Ruby code likes to access a C++ object, the first thing that happens is that a Ruby proxy object is created that is linked to the C++ object. That link can be unidirectional (the Ruby object knows about the C++ object, but the C++ object does not know about the Ruby object) or bidirectional (each know of each other). The kind of linking is important because a bidirectional link is stronger than a unidirectional link and allows lifetime tracking of the other object. For performance reason, not all objects implement the ability of bidirectional links, in particular not the ones that live in the layout database. That has certain consequences we will discuss in the lifetime management section below.

The Ruby proxy object serves as a connection point to the C++ object. It defines methods that correspond to methods in the C++ object. When one of these methods is called, their implementation collects the arguments of the method call and converts them to a binary representation that C++ understands. That process is usually called marshalling. Having done so, the execution continues in C++ space where the GSI framework will use the binary representation of the arguments to call the target method of the C++ object. After the call has returned the same happens on the way back, this time with the return value instead of the arguments. Having converted the return value back into Ruby objects the execution returns to the Ruby script.

The following image illustrates the relationship:



While that operation is simple in theory there are some pitfalls when implementing that scheme. One of them is the inherent compatibility issue between C++ and Ruby's lifetime management. In Ruby, the interpreter knows about all references to a Ruby object. When there are no more references to an object, the object is marked as "no longer used" and deleted. In other words: as long as any reference exists, the object is valid and a reference will never become invalid.

In C++, it is responsibility of the code to explicitly delete an object when it is no longer used. In other words: as long as any reference exists, the object is valid and a reference will never become invalid. Often there is a clear ownership: an object belongs to another object which controls the lifetime of the owned object (aggregation). That scheme is more efficient and predictable but it bears the danger of holding references to objects which are deleted already.

That raises the question how the lifetime of the Ruby proxy object is controlled and how the lifetime of the C++ object is related to that.

Lifetime management

RBA/GSI follows a simple principle that significantly simplifies the implementation: who created an object is responsible for cleaning it up. In other words: a ownership of an object is usually not transferred between C++ and Ruby space. Thus we have to consider two cases: The object is created in Ruby or the object is created in C++ code. Literally the object "lives" in Ruby space or in C++ space. In both cases, there is a pair of objects, but one of them is controlling the other.

Case 1: The object lives in Ruby space

When the object is created in Ruby, the Ruby proxy owns the C++ object and when the Ruby object goes out of scope, not only the Ruby object but also the C++ object is deleted. That means, that except if that case is handled by special measures, a reference to such an

object must not be stored in C++ space, because we will never know when Ruby will delete the object. A reference can be passed safely as an argument of a method however, provided the method does not store the reference somewhere.

But then: how do we then permanently store an object we have created in Ruby? The answer is simply by creating a copy on the C++ side. That is exactly what happens in that piece of code:

```
# creates an object that lives in Ruby space:
box = RBA::Box.new(0, 0, 10, 20)
# insert creates a copy of the box:
cell.shapes(layer).insert(box)
```

That is not an option for heavy objects such as layouts. If such objects need to be stored in C++ space, they are not created in Ruby code. Instead, several methods are provided to create objects that live in C++ space. For example a LayoutView object is not created in Ruby, but rather created inside the MainWindow object with create_layout. After that, the reference is obtained with MainWindow::current_view or MainWindow::view.

An exception to the lifetime control rule given above are Qt objects: a common pattern is to create Qt objects and add them to a container (i.e. widgets to a dialog). This implies a lifetime control transfer from Ruby to C++. RBA handles that case by explicitly transferring control when a QObject or one of the derived objects is created with a parent reference in Ruby code. Qt implements its own mechanism of controlling the lifetime which includes monitoring of the lifetime of child objects. This feature makes transferring the control feasible for these kind of objects. For some Qt methods which are know to transfer the ownership of an object, the ownership is transferred explicitly.

An object living in Ruby space can be explicitly deleted to free resources for example. For this, the "_destroy" method is provided. This method will only deleted the C++ object and not the Ruby object. However, the Ruby object will become invalid and calling a method on such an object will result in an error.

Case 2: the object lives in C++ space

In that case, the Ruby proxy object simply acts as a pointer to a C++ object. An issue arises when the Ruby object is still alive but the C++ object is deleted. In that case, the flavor of the link between the Ruby proxy and C++ object is important: if the link is bidirectional, the C++ object will inform the Ruby proxy that the reference will become invalid. The Ruby proxy will mark itself as being invalid and will block further calls to methods. Object supporting this reference binding are the API classes and "bigger" database objects such as Cell or Layout.

For example:

```
main_window = ... # the RBA::MainWindow object
# returns a reference to a RBA::LayoutView object living in C++ space:
view = main_window.current_view
# deletes all views and also the object we have a reference to
main_window.close_all
# this will fail, because the view is a Ruby proxy that knows that the C++ object
# has been deleted:
view.load_layout(...)
# You can check this by asking "_destroyed?". This will return "true":
view._destroyed?
```

For "lightweight" objects such as the geometry primitives (Box, Polygon etc.), the link is unidirectional and lifetime monitoring is not possible. This situation bears the danger of invalid references with fatal consequences if an attempt is made to call a method then. Fortunately this case is rare and usually mitigated by providing an object clone.

Calling "_destroy" on an object living in C++ space is not safe in general. In some cases, this can have fatal consequences (i.e. destroying the MainWindow object). An exception from that rule are Qt objects because Qt does lifetime monitoring internally and destroying an object from the outside (Ruby) is a valid operation in most cases (although there are exceptions).

Transfer Of Ownership And Object Lifetime

Some C++ methods accept pointers to objects and take over ownership over this object. This happens specifically inside the Qt methods. In that case, the Ruby object has to release ownership over that object. For example, "QApplication::postEvent" takes over the ownership over the event object passed to it and will finally destroy this object:

```
event = RBA::QKeyEvent::new
# takes over ownership over the event object:
RBA::QApplication::postEvent(RBA::MainWindow::instance, event)
...
```

```
# later on, when "event" goes out of scope, the GC will try to
# delete the QKeyEvent object and without further provisions, the
# application will crash!
```

Luckily, there are such provisions. The "postEvent" method is tagged specially, so the interpreter knows that it has to transfer ownership of the event object to Qt.

If that was not the case, one could use "_unmanage" to mark the event object no longer being managed by the script:

```
event = RBA::QKeyEvent::new
event._unmanage
# Now, somebody else is responsible for managing the object's lifetime
```

The reverse is true for methods delivering new objects which the Ruby interpreter is supposed to manage. For example "QLayout::takeAt" returns a free objects which the caller is responsible for deleting. Without further provisions this would lead to a memory leak, because Ruby does not delete the borrowed object:

```
layout = ... # A RBA::QLayout object
child = layout.takeAt(0)
# later on, when child goes out of scope, Ruby needs to delete the object.
# Here is does, because it knows that "takeAt" delivers a free object.
```

If the declaration of "takeAt" was not aware of the return mode, one could use "_manage" to mark the event object as to be managed by the script:

```
obj = createObjectForMe()
event._manage
# Now, when event goes out of scope, the object will be destroyed too.
```

Static And Local Methods, Access

Static C++ methods is simply implemented as class methods while local methods are implemented as instance methods.

RBA also supports binding of protected methods. A Ruby class derived from a C++ class exposed to Ruby can call those methods while code outside that class cannot access those methods. Public methods can be called from anywhere.

Data types, Arguments and Return values

Ruby and C++ feature different types of data. While in Ruby, a variable is of any type, in C++ a variable has a fixed type. This also is the case for arguments of methods and return values. A C++ method requires an argument to be of a certain type. In addition, C++ features pointers, references and a variety of containers. Therefore a mapping of Ruby types to C++ types is required. The following table summarizes the mapping for the simple types:

C++	Ruby	Comment
`(signed, unsigned) char, int, short, long, long long`	`FixNum`	
`float, double`	`Float`	
`const char *, std::string, QString, QByteArray`	`String`	KLayout uses UTF-8 encoding for std::string. Binary strings can be passed to and from QByteArray.
`bool`	`nil, true, false`	When passing a Ruby value to a bool parameter, the Ruby's nil and false values are converted to false. All other values are converted to true. This follows the usual Ruby semantics.
`void *`	`FixNum`	Passing pointers between Ruby and C++ is not possible. But often, a "void *" value is used as a handle or as an arbitrary value. The Ruby binding allows storing of such values as FixNum.

QVariant, tl::Variant (KLayout)	any	Any simple Ruby type that can be mapped to a C++ value can be stored in a QVariant. tl::Variant also supports a selection of complex types (i.e. RBA::Point, RBA::DPoint, RBA::Box, RBA::DBox etc.).

Arguments that expect objects of classes known to RBA can be passed references from Ruby objects. The linear containers (std::vector, QList etc.) are mapped to Ruby arrays. Their values can be any scalar type and objects of known classes. Nested arrays are not supported currently. Since the declaration is uniform in C++ (all members of an array must be of the same type), all members of a Ruby array must be convertible to the target type.

Pointers and references are a special topic for C++ to Ruby binding. Ruby does not have the concept of a reference. Instead, all object values are references by definition. RBA can convert between Ruby variables and pointers/references and also supports "out" parameters.

"void" as return value

While a "void" return value indicates "no return value" in C++, this concept is not common in Ruby. Ruby methods will always return the last value generated in a method.

For consistency, KLayout's Ruby binding returns "self" from methods in this case. This allows chaining of methods in many cases and fosters compact code:

```
// C++
class A {
public:
  virtual void f() { ... }
  virtual void g() { ... }
};

# Ruby
a = A::new.f.g
```

References and pointers to simple types (FixNum, Float, String)

Simple types can be passed as values to arguments expecting pointers and references. RBA will convert the value to a pointer to that value and pass that pointer to the method. Pointer arguments also support the "nil" value which is converted to a null pointer. Beware that not all C++ methods expecting a pointer argument are aware of null pointers and may have trouble digesting that value.

Often, (non-const) reference and pointer arguments are used as "out" parameters, i.e. the method alters the value of the memory location passed by the pointer value. That imposes a problem for the Ruby binding: since Ruby does not pass references for simple types, the value of a variable cannot be altered and the following code does not work as expected:

```
//  C++: use x as an "out" parameter:
void A::f(int &x) { x = 5; }

# Ruby:
x = 1
A::new.f(x)
# x is NOT 5!
```

RBA solves that problem by providing a "boxing" mechanism: a value is stored inside an object which is passed by reference. RBA provides the class Value for that purpose. This class serves as a container for any type and can be used to solve the "out" parameter problem this way:

```
//  C++: use x as an "out" parameter:
void A::f(int &x) { x = 5; }

# Ruby:
x = RBA::Value.new(1)
A::new.f(x)
# x.value is 5 now
```

The RBA::Value object has a attribute "value" which can hold any type. RBA will convert that member into the value required by the method's argument. The method will receive a pointer or reference to that value and can modify that memory location. After the method has returned, the modified value can be accessed by reading the "value" attribute.

Reference and pointer return values are simply converted to copies of the value.

Special pointers: const char *, void *

"const char *" pointers are mapped to Ruby strings. The same is true for "const unsigned char *". The non-const versions "char *" and "unsiged char *" are somewhat ambiguous and are mapped to strings currently.

"void *" is mapped to an integer value representing the address the pointer points at. Since it is not possible to address a value by a pointer in Ruby as well as getting the address of a value, there is noch much use for the "void *" values expect if those values are delivered and digested by C++ methods. This is the case for some "handle" values (i.e. Windows object handles) and some cases, where "void *" stands for "some arbitrary value which can be a pointer or an integer".

Pointers as arrays

In "C style" C++, pointers are sometimes used as the start position of an array and point to a number of items, not only one. Usually there is another parameter telling the number of items the pointer points to. Since there is no declaration for that kind of calling convention, the Ruby binding cannot support that case. Only "char *" and "const char *" are supported and it is assumed that arguments of these type expect a zero-terminated byte string in UTF-8 encoding.

Fortunately that case is rare. Specifically the Qt API uses references to QByteArray, QVector, QList and similar container classes which can be mapped to Ruby arrays.

References and pointers to complex types

References and pointers to complex types and objects of Ruby classes for which a C++ class exists are simply converted to pointers or references to the C++ class. Pointer arguments can also be passed "nil" which renders a null pointer. Again, not all method implementations may be prepared for that value and the application may crash in that case.

Hash arguments and return values

Associative containers (std::map, QHash etc.) are mapped to Ruby hashes. Unlike Ruby, C++ associative containers are strictly typed, so it's important to provide the right key and value pairs.

Default arguments

Some functions provide defaults for certain arguments. If these arguments are omitted, the default value is used instead.

Keyword arguments

Starting with version 3, Ruby supports "real" keyword arguments. Keyword arguments are supported for most methods with the exception of a few built-in ones such as "assign". Keyword arguments can be used when the other, non-optional arguments are given either by positional arguments or other keyword arguments.

Keyword arguments are somewhat more expressive and allow a shorter notation. For example, to instantiate a 45 degree rotation, you can write:

```
t = RBA::CplxTrans::new(rot: 45)
```

Implicit conversions

String arguments

If a method expects a string argument, other types are converted to strings using the "to_s" method. In Python, the equivalent is "str(...)".

Example:

```
# Also accepts a float value for the first string argument -
# it is converted to "2.5"
t = RBA::Text::new(2.5, RBA::Trans::new)
```

Conversion constructors

Conversion constructors are constructors that take an object of a different class and convert it to the target class. Conversion constructors are used implicitly in applicable cases to convert one type to the type requested by the argument.

For example, in the following code, the Region object's "+" operator is used. This expects a Region object as the second parameter, but as there is conversion constructor available which converts a Box to a Region, it is possible to use a Box directly:

```
r = RBA::Region::new(RBA::Box::new(0, 0, 1000, 2000))
r += RBA::Box::new(3000, 0, 4000, 2000)
```

Implicit constructor from lists

When an object is expected for an argument and a list is given, the object constructor is called with the arguments from the list. This specifically allows using size-2 lists instead of Point or Vector arguments. In Python, a "list" can also be a tuple.

In the following example, this mechanism is used for the polygon point list which is expected to be an array of Point objects, but can use size-2 arrays instead. Also, the "moved" method expects a Vector, but here as well, size-2 arrays can be used instead:

```
pts = [ [ 0, 0 ], [ 0, 1000 ], [ 1000, 0 ] ]
poly = RBA::Polygon::new(pts)
poly = poly.moved([ 100, 200 ])
```

Constness

C++ has the concept of "constness". That means that if a pointer or a reference to an object is declared "const", the object cannot be modified. Also, methods can be declared "const" meaning that such methods do no alter the (externally visible) state of an object. C++ ensures constness, because it only allows calling of const methods on const references of objects.

The concept of constness is part of the contract between a caller of a method and the method's implementation: if a method declares an argument to be a const reference or pointer, it tells the caller that it will not modify the object. Similar, returning a const reference is a safe way to expose internal objects because the implementor of the method can be sure that code outside a method will not modify the state of the object returned. Thus, constness is a vital part of a contract and somehow needs to be mapped in the Ruby binding.

Unfortunately that is not as easy as it may look. The problem is basically that Ruby does not have the concept of constness, but it has references. Remember that there is always a pair of Ruby/C++ objects. When a reference to a C++ object is returned into Ruby space, the Ruby counterpart of the object is created and a reference to that object is returned.

But where do we have to attach the constness of the reference? The answer is that there is no other place except the Ruby object. Hence, if a const reference is returned, a "const Ruby object" is created. That object will refuse to execute non-const methods. That way, the const semantics is maintained.

Trouble starts when another non-const reference is returned to the same object. In that case, the Ruby object needs to be reused but this time with non-const semantics. That is a contradiction to the previous const state. RBA solves this issue by switching the object to non-const state in that case and will allow to call non-const methods after that.

In other words: constness is part of the object identity in Ruby and it can change. That actually makes some sense: when I obtained a const reference there may be another way to obtain a non-const reference. Once I have a non-const reference I can modify the object which also is behind the const reference. Thus keeping a const reference is no longer a safety feature and the const reference can be dropped.

To avoid lifetime issues, RBA does not work with references a lot. Objects returned by a const reference are always copied. Only const pointers are kept as const object references in Ruby.

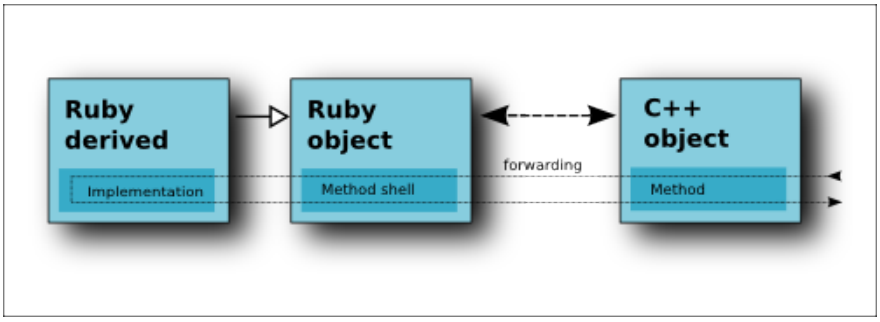
Reimplementing Virtual Methods

The Ruby binding supports reimplementation of virtual C++ methods in derived classes. This works as expected:

```
// C++
class A {
public:
  virtual void f() { }
};
```

```
# Ruby
class B < A
  def f
    # f is called when A::f is called on the C++ side
  end
end
```

Virtual methods are often used as callbacks and provide a reverse call path from C++ to Ruby:



The parameters of the Ruby implementation must match the parameters of the C++ method. Mapping of Ruby to C++ types applies to the Ruby method arguments the same way than for return values of ordinary methods. Virtual functions can also return values. In that case the same mapping rules than for ordinary method arguments apply.

Iterators

A special feature of KLayout's Ruby binding are iterators. C++ iterators are mapped to Ruby iterators. For example:

```
// C++
class A {
public:
  //  begin()..end() are mapped to the "each" method in Ruby:
  iterator begin();
  iterator end();
};

# Ruby
a = A::new
a.each { |i| ... }
```

If no block is given, an Enumerator object is created. Enumerators are a Ruby feature. Enumerators support many convenient methods like sort, inject, collect, select etc. Here is an example:

```
# turns all elements returned by the iterator into strings and sorts them
sorted = a.each.collect(&:to_s).sort
```

Iterators match very well between C++ and Ruby so there are no real issues here. The return type of the iterators is mapped to Ruby's block arguments using the same rules than for values returned from C++ methods.

Exceptions

Raising an error in Ruby is a valid way to terminate the execution of the method. A Ruby error is mapped to a C++ exception which usually is caught in the C++ code and handled properly. There are some cases, where raising an exception can crash the application. That is the case in particular in event handlers of Qt objects. Usually, raising an exception is safe.

Events

Events are a special feature of KLayout's Ruby binding. Events are similar to reimplementations of virtual functions except that no derived class is required and the call is handled by a Ruby block. Events can have return values but using "return" inside a block does not have the expected effect. Instead, the value of the last expression in the block is used. That is a feature of Ruby, not a speciality of RBA.

There is always one receiver for an event. If a new block is assigned to an event, the old block will no longer be called. Here is an example of using events:

```
// C++
class A {
public:
  //  e is an event with an integer argument
  void f(i) { e(i); }
};

# Ruby
a = A::new
a.e { |i| puts i; }
a.f(15)  # calls the block attached to e with the argument 15
```

Events are extensively used for an alternative to Qt slots. The Qt binding of KLayout maps every signal to an event. That means, that it is possible to connect a block to a Qt signal directly at the sender object without having to create a receiver. For example:

```
# Ruby
b = RBA::QPushButton::new
# print a message, when the button is clicked
b.clicked { puts "Ouch." }
```

There is one significant difference between Qt signals and events: A Qt signal can have many receivers while an event always has one block which is executed when the signal is emitted. Connecting signals and slots still is supported with the "connect" method, but it is not possible to define slots on Ruby methods. The events fill that gap and, in the authors opinion, in a much more convenient way.

3.8. Coding PCells In Ruby

A good starting point for Ruby PCells is the PCell sample. Create a macro in the macro development IDE (use the "+" button) and choose "PCell sample" from the templates.

The Sample

We'll do a code walk through that sample here and explain the concepts while doing so. Here is the complete sample:

```
# Sample PCell
#
# This sample PCell implements a library called "MyLib" with a single PCell that
# draws a circle. It demonstrates the basic implementation techniques for a PCell
# and how to use the "guiding shape" feature to implement a handle for the circle
# radius.
#
# NOTE: after changing the code, the macro needs to be rerun to install the new
# implementation. The macro is also set to "auto run" to install the PCell
# when KLayout is run.

module MyLib

  include RBA

  # Remove any definition of our classes (this helps when
  # reexecuting this code after a change has been applied)
  MyLib.constants.member?(:Circle) && remove_const(:Circle)
  MyLib.constants.member?(:MyLib) && remove_const(:MyLib)

  # The PCell declaration for the circle
  class Circle < PCellDeclarationHelper

    include RBA

    def initialize

      # Important: initialize the super class
      super

      # declare the parameters
      param(:l, TypeLayer, "Layer", :default => LayerInfo::new(1, 0))
      param(:s, TypeShape, "", :default => DPoint::new(0, 0))
      param(:r, TypeDouble, "Radius", :default => 0.1)
      param(:n, TypeInt, "Number of points", :default => 64)
      # this hidden parameter is used to determine whether the radius has changed
      # or the "s" handle has been moved
      param(:ru, TypeDouble, "Radius", :default => 0.0, :hidden => true)

    end

    def display_text_impl
      # Provide a descriptive text for the cell
      "Circle(L=#{l.to_s},R=#{'%.3f' % r.to_f})"
    end

    def coerce_parameters_impl

      # We employ coerce_parameters_impl to decide whether the handle or the
      # numeric parameter has changed (by comparing against the effective
      # radius ru) and set ru to the effective radius. We also update the
      # numerical value or the shape, depending on which on has not changed.
      rs = nil
      if s.is_a?(DPoint)
        # compute distance in micron
        rs = s.distance(DPoint::new(0, 0))
```

```
      end
      if rs && (r-ru).abs < 1e-6
        set_ru rs
        set_r rs
      else
        set_ru r
        set_s DPoint::new(-r, 0)
      end

      # n must be larger or equal than 4
      n > 4 || (set_n 4)

    end

    def can_create_from_shape_impl
      # Implement the "Create PCell from shape" protocol: we can use any shape which
      # has a finite bounding box
      shape.is_box? || shape.is_polygon? || shape.is_path?
    end

    def parameters_from_shape_impl
      # Implement the "Create PCell from shape" protocol: we set r and l from the shape's
      # bounding box width and layer
      set_r shape.bbox.width * layout.dbu / 2
      set_l layout.get_info(layer)
    end

    def transformation_from_shape_impl
      # Implement the "Create PCell from shape" protocol: we use the center of the shape's
      # bounding box to determine the transformation
      Trans.new(shape.bbox.center)
    end

    def produce_impl

      # This is the main part of the implementation: create the layout

      # fetch the parameters
      ru_dbu = ru / layout.dbu

      # compute the circle
      pts = []
      da = Math::PI * 2 / n
      n.times do |i|
        pts.push(Point.from_dpoint(DPoint.new(ru_dbu * Math::cos(i * da), ru_dbu * Math::sin(i * da))))
      end

      # create the shape
      cell.shapes(l_layer).insert(Polygon.new(pts))

    end

  end

  # The library where we will put the PCell into
  class MyLib < Library

    def initialize

      # Set the description
      self.description = "My First Library"

      # Create the PCell declarations
      layout.register_pcell("Circle", Circle::new)
      # That would be the place to put in more PCells ...

      # Register us with the name "MyLib".
      # If a library with that name already existed, it will be replaced then.
```

```
      register("MyLib")

    end

  end

  # Instantiate and register the library
  MyLib::new

end
```

Preamble

The first important concepts are PCell class and library. A PCell is provided by implementing a certain class and providing the functionality of the PCell through various methods. In fact there are only three methods which must be implemented. In the sample we use PCellDeclarationHelper as the base class for our PCell. This is a convenience wrapper around the basic interface, PCellDeclaration. Since that interface is too much "C++"-like and is somewhat tedious to use, the PCellDeclarationHelper is the recommended starting point.

Using the same concept, a library is an object derived from the Library class. It is basically a container for PCells and static layout cells. A library has to be initialized (most conveniently in the constructor), registered and initialized once. That makes the library available to the system and it can be used in layouts.

Please note, that the sample PCell is configured for auto-run. This way, the library is installed when KLayout starts and before any layouts are loaded. That way, the library is available for layouts read from the command-line for example.

Let's now start with our code walk:

```
module MyLib

  include RBA

  # Remove any definition of our classes (this helps when
  # reexecuting this code after a change has been applied)
  MyLib.constants.member?(:Circle) && remove_const(:Circle)
  MyLib.constants.member?(:MyLib) && remove_const(:MyLib)
```

It is recommended to put the library code into a separate module. That allows mixing in other modules (in that case RBA) without affecting the main module. The second recommendation is to remove classes which are already defined with the names we are going to create. While developing a PCell is it necessary to frequently rerun the script to register the new version of the library and PCell. If we do not remove the existing class, Ruby will refuse to reopen a class for example if we change the super class or methods we have deleted will still remain. That is avoided by removing the classes before the create them again. In Ruby, a class can be removed by removing the constant with the class name. Note the way, the script checks whether a class is defined by using "member?" on the list of constants. This method should be preferred over "const_defined?" which behaves differently on Ruby 1.8 and Ruby 1.9.

The PCell Class

First we define a PCell class derived from PCellDeclarationHelper. This is the most convenient way to declare a PCell:

```
  # The PCell declaration for the circle
  class Circle < PCellDeclarationHelper

    include RBA
```

Again we include RBA which allows us to use RBA classes inside the PCell without having to write "RBA::" in front of the class names.

The initialization of the object is already a very important step. First, it must initialize the super class. Then it has to declare the PCell parameters. Each PCell has a set of parameters that define the appearance of the PCell. Parameters have a symbolic name, a type, a description and optionally a default value and further attributes. The name is important because it identifies the parameter throughout the system and in layout files as well. It should not be changed. The description is an arbitrary string and can be changed or localized.

Parameters are declared using the "param" method of PCellDeclarationHelper:

```
    def initialize
```

```
    # Important: initialize the super class
    super

    # declare the parameters
    param(:l, TypeLayer, "Layer", :default => LayerInfo::new(1, 0))
    param(:s, TypeShape, "", :default => DPoint::new(0, 0))
    param(:r, TypeDouble, "Radius", :default => 0.1)
    param(:n, TypeInt, "Number of points", :default => 64)
    # this hidden parameter is used to determine whether the radius has changed
    # or the "s" handle has been moved
    param(:ru, TypeDouble, "Radius", :default => 0.0, :hidden => true)

  end
```

In that sample we declared a PCell parameter "l" with type "TypeLayer" which indicates that this is a layer in the layout. "s" is a parameter shape represents the handle and is a shape. A shape is either of type DBox, DText, DPath, DPolygon or DPoint. Shape parameters implement the "guiding shape feature" that KLayout offers to manipulate that parameter graphically. "r" and "n" are simple numerical parameters. All parameters have default values which are set with the "default" symbolic parameter. As a layer, "l" must have a LayerInfo value. "s" is a DPoint which reflects the handle. Since default values not only preset the parameters to a reasonable value but also define the subtype of a parameter (here the DPoint shape), providing a default is strongly recommended. As shapes need to be independent from the database unit for portability, they are expressed in micron units. Hence the use of the "D" forms (DPoint etc.).

"ru" is a special parameter. Because we have two ways to modify the radius (the handle and the numerical value), it is used as a shadow parameter do determine which one of these two values has changed. Depending on that information, either the handle or the radius is updated. Because this parameter should not be shown in the parameter page, it is marked "hidden".

There are some more options for parameters. See the documentation of PCellDeclarationHelper for more details about the further attributes.

The parameter declaration will create accessor methods for each parameter. These accessor methods can be used to get and set the current value of the parameter inside the production method and other methods. For that, it will use the symbolic name of the parameter. The setter is called "set_x" (where x is the parameter name). Although Ruby would allow using "x=" to mimic an assignment, this option leads to some confusion with definition of local variables and was not considered here. The following methods are created in the sample:

- `l, set_l, l_layer`: getter and setter for the current value of "l". l_layer is the layer index in the context of the PCell production method. The layer index can be used to access the layer in the layout or cell.
- `s, set_s`: getter and setter for the current value of "s".
- `r, set_r, n, set_n, ru, set_ru`: same for "r", "n" and "ru".

After the PCell initialization is finished, we can start with the production code. These are the methods that KLayout will call on certain opportunities. The first method that a PCell must implement is the display text callback:

```
  def display_text_impl
    # Provide a descriptive text for the cell
    "Circle(L=#{l.to_s},R=#{'%.3f' % r.to_f})"
  end
```

KLayout will call this method to fetch a formatted string that represents the content of a PCell. This text is used for the cell tree and cell box labels. To avoid confusion, it should start with the name of the PCell. The bracket notation is not mandatory, but it's always a good idea to follow some common style. The information delivered by this method should be short but contain enough information so that a PCell variant can be distinguished from its sibling.

The next method is called whenever something on the parameters has changed. This method allows to adjust the parameters so that they obey certain limitations. It can also raise exceptions for invalid parameter combinations. In our case we use this method to adjust the handle or the numeric radius to the effective value. We also enforce a minimum number of vertex counts for the resulting polygon. Implementing this method in general is optional. By default, no modification of the parameters is done:

```
  def coerce_parameters_impl

    # We employ coerce_parameters_impl to decide whether the handle or the
    # numeric parameter has changed (by comparing against the effective
    # radius ru) and set ru to the effective radius. We also update the
```

```
    # numerical value or the shape, depending on which on has not changed.
    rs = nil
    if s.is_a?(DPoint)
      # compute distance in micron
      rs = s.distance(DPoint::new(0, 0))
    end
    if rs && (r-ru).abs < 1e-6
      set_ru rs
      set_r rs
    else
      set_ru r
      set_s DPoint::new(-r, 0)
    end

    # n must be larger or equal than 4
    n > 4 || (set_n 4)

  end
```

The implementation of the following three methods is optional: they are used to implement the "PCell from shape" protocol. If "Create PCell from shape" is selected in KLayout's Edit menu, it will call "can_create_from_shape_impl" for each known PCell. This method will be given the shape, layout and layer. If this method responds with "true", KLayout offers this PCell as a conversion target in the list. When this PCell has been selected, KLayout calls "parameters_from_shape_impl" and "transformation_from_shape_impl" to obtain the initial parameters and the initial transformation for the new PCell created from that shape. "parameter_from_shape_impl" will use the default values for all parameters unless they are set with the respective setters in the implementation body.

```
  def can_create_from_shape_impl
    # Implement the "Create PCell from shape" protocol: we can use any shape which
    # has a finite bounding box
    shape.is_box? || shape.is_polygon? || shape.is_path?
  end

  def parameters_from_shape_impl
    # Implement the "Create PCell from shape" protocol: we set r and l from the shape's
    # bounding box width and layer
    set_r shape.bbox.width * layout.dbu / 2
    set_l layout.get_info(layer)
  end

  def transformation_from_shape_impl
    # Implement the "Create PCell from shape" protocol: we use the center of the shape's
    # bounding box to determine the transformation
    Trans.new(shape.bbox.center)
  end
```

The most important method is "produce_impl" which actually creates the layout. For that, it can use all methods of Layout and Cell and most other RBA classes. It can even create instances. Although that is possible, it is not recommended to create cells in the production code. This would pretty much degrade performance and lead to a confusing variety of cells. It is possible to use boolean operations by using the methods of EdgeProcessor for example. Some care must be taken to avoid interaction with the user interface, in particular calling methods of LayoutView and MainWindow should be avoided.

The actual layout of the PCell is cached and the production code is called only when the PCell parameters have changed. However, to reduce the risk of performance degradation, the method should run quickly and not spend too much time in long loops or huge data sets.

```
  def produce_impl

    # This is the main part of the implementation: create the layout

    # fetch the parameters
    ru_dbu = ru / layout.dbu

    # compute the circle
    pts = []
    da = Math::PI * 2 / n
    n.times do |i|
```

```
        pts.push(Point.from_dpoint(DPoint.new(ru_dbu * Math::cos(i * da), ru_dbu * Math::sin(i * da))))
      end

      # create the shape
      cell.shapes(l_layer).insert(Polygon.new(pts))

    end

  end
```

Of course, more than one PCell class can be declared. Each PCell type must have an own implementation class which we will use later to create the PCells from.

The Library

The library is the container for the PCells. All important code is packed into the constructor of the library.

```
  # The library where we will put the PCell into
  class MyLib < Library

    def initialize

      # Set the description
      self.description = "My First Library"

      # Create the PCell declarations
      layout.register_pcell("Circle", Circle::new)
      # That would be the place to put in more PCells ...

      # Register us with the name "MyLib".
      # If a library with that name already existed, it will be replaced then.
      register("MyLib")

    end

  end
```

First, a library needs a description that we set with the description setter. Then, we instantiate all PCell classes once and register that instance in the library space.

The library is basically an ordinary Layout object that we can access through the "layout" method. The library can consist of more that PCells - all cells that we put into the layout will become available as library components (more precisely: all top cells). We could use RBA::Layout::read for example to feed the layout with cells from a file.

At the end of the constructor we register our instance inside the system with the given name. To avoid confusion, it is recommended to use the same name for the class and the library.

Finally we only need to instantiate the library:

```
  # Instantiate and register the library
  MyLib::new
```

This line of code will instantiate the library and, through the constructor, instantiate the PCells and register the library. We are done now and can use the library and our PCell.

Debugging The Code

When you have modified the code, you need to rerun the script. That will create the classes again and re-register the PCells and the Library with the new implementation. PCells already living in the layout will be migrated to the new implementation by mapping their parameters by their symbolic names.

The PCell code can be debugged with KLayout's built-in Ruby debugger. If the macro development IDE window is open, just load the PCell code and set a breakpoint. When KLayout calls the PCell implementation, the breakpoint will be triggered. Local variables can be inspected

and modified in the console for example. Single-stepping is supported as well. If the execution is stopped, KLayout will finish the operation with some error message.

It is also possible to print output to the console if the macro development IDE is open. Just use the methods of stdout that Ruby offers or simply "puts".

Please note that while the macro development IDE is opened, macro execution is considerably slower than usually, because the IDE will plug itself into the Ruby interpreter and trace the execution. When the IDE window is closed, Ruby runs at full speed. While in a breakpoint, KLayout's main window is only half alive. Only the IDE is active and the main window will not even repaint correctly. This prevents possible interactions with the executed code.

3.9. The Qt Binding

Starting with 0.22 comes with a large set of Qt classes available through the Ruby binding mechanism. This allows integration of Qt user interfaces with Ruby scripts and to use the Qt API, for example the network or SQL classes. To use the Qt bindings, KLayout must be compiled with Qt binding, i.e.

```
build.sh -with-qtbinding ...
```

The API provided covers the functionality of a certain Qt version. Currently this is Qt 4.6 and Qt 5.5. The API covers the following Qt 4 and Qt 5 modules:

- QtCore: the basic Qt API
- QtGui: the user interface widgets and supporting classes
- QtXml: support for XML
- QtXmlPatterns (Qt5): XML schema and queries
- QtSql: database support
- QtNetwork: various network protocols and supporting classes
- QtDesigner: dynamically load designer files (.ui)
- QtUiTools: dynamically load designer files (.ui)
- QtMultimedia (Qt5): multimedia support
- QtPrintSupport (Qt5): print support
- QtSvg (Qt5): SVG implementation

This article covers the use of the Qt API and special topics related to that. It is recommended to read the article about the Ruby binding (The Ruby Language Binding) for a deeper understanding of the mapping of the Qt API to Ruby.

There is some overlap with the "qtruby" project. This project also makes the Qt API available for Ruby. The approach of "qtruby" is similar. Yet there are some differences, in particular the event feature of KLayout, which allows a convient binding of code blocks to signals.

A First Sample

This is a first sample of how to use the Qt API. To run that sample, open the Macro Development IDE from the "Macros" menu. Add a new macro with the "+" button in the left top tool bar. Choose "General KLayout Macro" from the "General" group as the template. Paste the code above into the macro and run the macro with F5.

```
module MyMacro

  include RBA

  dialog = QDialog::new(Application.instance.main_window)
  dialog.windowTitle = "My Dialog"

  layout = QHBoxLayout::new(dialog)
  dialog.setLayout(layout)

  button = QPushButton.new(dialog)
  layout.addWidget(button)
  button.text = "Click Me"
  button.clicked do
    QMessageBox::information(dialog, "Message", "I was clicked!")
  end
```

```
    dialog.exec

  end
```

The sample creates a `QDialog` with a layout and a button in it. When the button is clicked, a message box appears. This code demonstrates some features of the Ruby binding of the Qt library. For example, the button's text is set with an attribute assignment ("`button.text = ...`") rather than a method call ("`button.setText(...)`").

A noteworthy feature is the event binding which allows associating code blocks with signals. In pure Qt, the "clicked" signal of the button would have to be connected to a slot. This is not possible without creating a receiver object. With events, that receiver is created internally and a code block can be attached to the signal directly:

```
button.clicked do
  QMessageBox::information(dialog, "Message", "I was clicked!")
end
```

That binding includes the ability to receive signal arguments through block arguments.

Binding Details

Given the rules stated in the general Ruby binding documentation (The Ruby Language Binding) the ruby versions of most methods can be derived readily. There are some exceptions however that we will cover here.

First, the C++ to Ruby binding lacks some features that are required for some methods. Those methods cannot be bound. Specifically that concerns:

- Methods that return or require unsupported containers
- Methods that return or require function pointers
- Methods that return or require pointers to pointers or references to pointers
- Methods that require pointers to pass arrays (i.e. `QPolygon::putPoints`)
- Methods that return or require objects which are not available in Ruby for some reason (i.e. `QGenericArgument` or some template classes like `GenericMatrix`)
- Template members
- Methods which are not available on one of the platforms (i.e. `QPrinter::printerSelectionOption`, which is not available on Windows)
- Some operators, like the cast operators
- Methods that require "char *" arguments which will be manipulated (i.e. `QFile::readLineData`)
- Methods that require C++ resources (i.e FILE arguments)
- Methods for which an alternative exists which cannot be distiguished (i.e. variants acceptings `QByteArray` and `QString` - only the `QString` variant is bound in that case)
- Methods and operators which requires a typed argument as disambiguator (i.e. `operator>>`, where the variant is selected by examining the type of the argument. For that case, alternatives exists which explicitly state the type in the name, for example "`read_i32`")

Some template types are made available to Ruby. In particular that is valid for some `QPair` specializations. For example "`QPair<double, double>`" is available as "`QDoublePair`".

The naming of some methods has been aligned to Ruby:

- `isX` is available as `isX?`, i.e. `QAction::isSeparator?`

- `setX` is available as `x=`, i.e. `QAction::iconText=`

In those cases, the original declaration is still available also.

The inheritance hierarchy of classes is mapped to Ruby in most cases. Sometimes that is not possible. For example, if the base class is a template (i.e. `QPolygon`, where the base class is a `QVector<QPoint>`). In that case, methods are provided that implement the features from the base class in the derived class.

Operators are bound to Ruby operators where that makes sense. For example, for `QPoint` the operators are available as expected ("==", "+" etc.).

"`destroy`" and "`create`" have been renamed to "`qt_destroy`" and "`qt_create`" to avoid name clashes with identical methods inherited from the RBA binding. "`destroy`" and "`create`" are standard methods which KLayout's Ruby binding defines for every object exposed to Ruby.

When a Qt object is created in Ruby space with a parent, its ownership is passed to the parent. It is safe however to keep a reference to that object, because KLayout's Ruby binding employs a special Ruby class internally (a proxy) which keeps track of the lifetime of the Qt object. If the parent is destroyed, the Qt object is destroyed as well. The internal Qt object will be notified and the reference to the Qt object will be invalidated. As a consequence, the Ruby proxy will refuse to execute methods on that object.

Destroying a Qt object can be necessary for example to free resources. To perform the equivalent of the C++ delete operator, use the "`destroy`" method that comes with every class exposed to Ruby. After an object is destroyed, the Ruby part of the binding still persists until all references to that object are removed. However, it is no longer possible to call methods on these objects.

The Qt binding significantly benefits from the dynamic binding of C++ objects. If a C++ pointer is returned, this pointer often is a pointer to a base class. Behind the pointer often is an object of a derived class. C++ allows calling of base class methods on that pointer, but not methods of the derived class - the identity of the object is reduced to the base class.

For example, if a method returns a `QWidget` pointer for a `QPushButton`, it is not possible to directly set the buttons text, because the method required for that is not part of the `QWidget` interface. In C++ one would dynamic_cast the pointer to `QWidget` and set the text then.

The Ruby binding automatically upcasts the pointer to the actual object, so the value returned has the real object's identity. In that case, delivering a `QWidget` would render a Ruby object that has a `QPushButton` identity and it's possible to set the text immediately. If the object was not a `QPushButton`, an error would be issued when an attempt is made to call a `QPushButton` method.

Enums

Enum types are available as classes to give them a specific context. Since Ruby does not allow declaration of classes within classes, enums declared inside a class must be declared as separate classes outside that class. The relationship is indicated by the Enum's class name. For example, `QMessageBox::Icon` (C++) is available as the Ruby class `QMessageBox_Icon`. The enum values are defined as constants within that class and the enclosing class. For example `QMessageBox::Critical` which is a value for `QMessageBox::Icon` is available as `QMessageBox_Icon::Critical` and `QMessageBox::Critical` in Ruby.

Starting with version 0.24, the `QFlags` template is supported as a separate class. The name of the class indicates the relationship to the enum class. For example, `QFlags<QMessageBox::Icon>` is available as `QMessageBox_QFlags_Icon`. Enum classes are derived from their respective flags class, so they can serve to initialize arguments expecting flags. It's hardly required to operate with the flags classes directly, since they are created automatically when joining enum's with the "or" (`|`) operator:

```
QMessageBox::Ok                        # A QMessageBox_StandardButton object
QMessageBox::Ok | QMessageBox::Cancel  # A QMessageBox_QFlags_StandardButton object
```

With these definitions, the following is allowed:

```
QMessageBox::information(parent, title, text, QMessageBox::Ok | QMessageBox::Cancel)
QMessageBox::information(parent, title, text, QMessageBox::Ok)
```

Using the designer

It is possible to load a dialog from a UI designer (.ui) file. Have a look at the following sample:

```
module MyMacro

  include RBA
```

```
    ui_file = QFile::new(QFileInfo::new($0).dir.filePath("MyDialog.ui"))
    ui_file.open(QIODevice::ReadOnly)
    dialog = QFormBuilder::new.load(ui_file, Application::instance.main_window)
    ui_file.close

    def dialog.setup
      button.clicked do
        slider.value = (slider.value + 1) % 100
      end
    end

    dialog.setup
    dialog.exec

  end
```

This sample tries to locate a designer file called "`MyDialog.ui`" relative to the macro's path (in `$0`). It uses the QFormBuilder class to load and create the dialog. In that sample, "`MyDialog`" defines a dialog with two widgets: a `QPushButton` ("button") and a `QSlider` ("slider"). Because of the dynamic binding in Ruby, "dialog" will already have the correct class and we don't have to cast the pointer delivered by `QFormBuilder::load` before we can call "exec".

This sample exploits a nice Ruby feature: in Ruby it is possible to dynamically add methods to an instance. This allows extending the `QDialog` object we got from `QFormLoader` by custom code. In our case we add a "setup" method. This method installs the custom logic of the dialog. It makes use of a convenience feature implemented in `QObject`'s Ruby binding: all named child objects of an object are available through accessor methods. Therefore we can access "button" and "slider" by their name to install an event handler that updates the slider value each time the button is clicked.

After calling setup on the dialog we have initialized it and we can show it with "exec".

Behind The Scenes

The mechanism behind the Ruby binding a based on the RBA/GSI framework of KLayout. In order to be able to derive from existing classes, that framework needs to add a kind of interfacing class atop of existing classes. Thus, the framework exposes every Qt class in two ways:

- Directly, without the ability to reimplement virtual methods. This is the way, existing Qt objects are addressed. The Ruby classes for that case are called "X_Native" where "X" is the name of the Qt class (for example, "`QObject_Native`"). When you receive a reference to a Qt object created by C++ code, this reference will have the native class type.
- Indirectly through an interface class. This is the object created when you instantiate a Qt object in Ruby. All virtual methods will be rerouted to the Ruby dispatcher. That allows to reimplement every virtual method, but also adds some overhead to every virtual method call. The name of these classes is identical to the name of the Qt class. In the swig tool, those classes are called "director classes".

The difference between the native and interface classes is important if you test the type of a object. The difference between both cases is the scope. The native classes will always match. The interface classes will only match if the object was created by Ruby code:

```
b = dialog.button
# this will not render true, if the button was created by QFormBuilder for example
b.is_a?(QPushButton)
# this is correct:
b.is_a?(QPushButton_Native)
```

To avoid confusion, the native classes do not appear in the documentation. They would just add another level of inheritance without providing additional methods.

4. Class Index

Per-Module documentation:

- Core Module db
- Core Module lay
- Core Module rdb
- Core Module tl

KLayout classes

AbsoluteProgress	tl	A progress reporter counting progress in absolute units
AbstractMenu	lay	An abstraction for the application menus
AbstractProgress	tl	The abstract progress reporter
Action	lay	The abstraction for an action (i.e. used inside menus)
Annotation	lay	A layout annotation (i.e. ruler)
Application	lay	The application object
BitmapBuffer	lay	A simplistic pixel buffer representing monochrome image
Box	db	A box class with integer coordinates
BrowserDialog	lay	A HTML display and browser dialog
BrowserPanel	lay	A HTML display and browser widget
BrowserSource	lay	The BrowserDialog's source for "int" URL's
ButtonState	lay	The namespace for the button state flags in the mouse events of the Plugin class.
Cell	db	A cell
CellInstArray	db	A single or array cell instance
CellMapping	db	A cell mapping (source to target layout)
CellView	lay	A class describing what is shown inside a layout view
Circuit	db	Circuits are the basic building blocks of the netlist
CompoundRegionOperationNode	db	A base class for compound DRC operations
CompoundRegionOperationNode::Geometri	db	This class represents the CompoundRegionOperationNode::GeometricalOp enum
CompoundRegionOperationNode::LogicalOp	db	This class represents the CompoundRegionOperationNode::LogicalOp enum

CompoundRegionOperationNode::Paramete	db	This class represents the parameter type enum used in \CompoundRegionOperationNode#new_bbox_filter
CompoundRegionOperationNode::RatioParamdbterType		This class represents the parameter type enum used in \CompoundRegionOperationNode#new_ratio_filter
CompoundRegionOperationNode::ResultTy	db	This class represents the CompoundRegionOperationNode::ResultType enum
Connectivity	db	This class specifies connections between different layers.
CplxTrans	db	A complex transformation
Cursor	lay	The namespace for the cursor constants
D25View	lay	The 2.5d View Dialog
DBox	db	A box class with floating-point coordinates
DCellInstArray	db	A single or array cell instance in micrometer units
DCplxTrans	db	A complex transformation
DEdge	db	An edge class
DEdgePair	db	An edge pair (a pair of two edges)
DPath	db	A path class
DPoint	db	A point class with double (floating-point) coordinates
DPolygon	db	A polygon class
DSimplePolygon	db	A simple polygon class
DText	db	A text object
DTrans	db	A simple transformation
DVector	db	A vector class with double (floating-point) coordinates
DeepShapeStore	db	An opaque layout heap for the deep region processor
Device	db	A device inside a circuit.
DeviceAbstract	db	A geometrical device abstract
DeviceAbstractRef	db	Describes an additional device abstract reference for combined devices.
DeviceClass	db	A class describing a specific type of device.
DeviceClassBJT3Transistor	db	A device class for a bipolar transistor.
DeviceClassBJT4Transistor	db	A device class for a 4-terminal bipolar transistor.
DeviceClassCapacitor	db	A device class for a capacitor.

DeviceClassCapacitorWithBulk	db	A device class for a capacitor with a bulk terminal (substrate, well).
DeviceClassDiode	db	A device class for a diode.
DeviceClassFactory	db	A factory for creating specific device classes for the standard device extractors
DeviceClassInductor	db	A device class for an inductor.
DeviceClassMOS3Transistor	db	A device class for a 3-terminal MOS transistor.
DeviceClassMOS4Transistor	db	A device class for a 4-terminal MOS transistor.
DeviceClassResistor	db	A device class for a resistor.
DeviceClassResistorWithBulk	db	A device class for a resistor with a bulk terminal (substrate, well).
DeviceExtractorBJT3Transistor	db	A device extractor for a bipolar transistor (BJT)
DeviceExtractorBJT4Transistor	db	A device extractor for a four-terminal bipolar transistor (BJT)
DeviceExtractorBase	db	The base class for all device extractors.
DeviceExtractorCapacitor	db	A device extractor for a two-terminal capacitor
DeviceExtractorCapacitorWithBulk	db	A device extractor for a capacitor with a bulk terminal
DeviceExtractorDiode	db	A device extractor for a planar diode
DeviceExtractorMOS3Transistor	db	A device extractor for a three-terminal MOS transistor
DeviceExtractorMOS4Transistor	db	A device extractor for a four-terminal MOS transistor
DeviceExtractorResistor	db	A device extractor for a two-terminal resistor
DeviceExtractorResistorWithBulk	db	A device extractor for a resistor with a bulk terminal
DeviceParameterDefinition	db	A parameter descriptor
DeviceReconnectedTerminal	db	Describes a terminal rerouting in combined devices.
DeviceTerminalDefinition	db	A terminal descriptor
Dispatcher	lay	Root of the configuration space in the plugin context and menu dispatcher
Edge	db	An edge class
EdgeFilter	db	A generic edge filter adaptor
EdgeMode	db	This class represents the edge mode type for \Region#edges.
EdgeOperator	db	A generic edge-to-polygon operator
EdgePair	db	An edge pair (a pair of two edges)
EdgePairFilter	db	A generic edge pair filter adaptor

EdgePairOperator	db	A generic edge-pair operator
EdgePairToEdgeOperator	db	A generic edge-pair-to-edge operator
EdgePairToPolygonOperator	db	A generic edge-pair-to-polygon operator
EdgePairs	db	EdgePairs (a collection of edge pairs)
EdgeProcessor	db	The edge processor (boolean, sizing, merge)
EdgeToEdgePairOperator	db	A generic edge-to-edge-pair operator
EdgeToPolygonOperator	db	A generic edge-to-polygon operator
Edges	db	A collection of edges (Not necessarily describing closed contours)
Edges::EdgeType	db	This enum specifies the edge type for edge angle filters.
EmptyClass	tl	
EqualDeviceParameters	db	A device parameter equality comparer.
Executable	tl	A generic executable object
Expression	tl	Evaluation of Expressions
ExpressionContext	tl	Represents the context of an expression evaluation
FileDialog	lay	Various methods to request a file name
GenericDeviceCombiner	db	A class implementing the combination of two devices (parallel or serial mode).
GenericDeviceExtractor	db	The basic class for implementing custom device extractors.
GenericDeviceParameterCompare	db	A class implementing the comparison of device parameters.
GenericNetlistCompareLogger	db	An event receiver for the netlist compare feature.
GlobPattern	tl	A glob pattern matcher
HAlign	db	This class represents the horizontal alignment modes.
HelpDialog	lay	The help dialog
HelpSource	lay	A BrowserSource implementation delivering the help text for the help dialog
ICplxTrans	db	A complex transformation
IMatrix2d	db	A 2d matrix object used mainly for representing rotation and shear transformations (integer coordinate version).
IMatrix3d	db	A 3d matrix object used mainly for representing rotation, shear, displacement and perspective transformations (integer coordinate version).

Image	lay	An image to be stored as a layout annotation
ImageDataMapping	lay	A structure describing the data mapping of an image object
InputDialog	lay	Various methods to open a dialog requesting data entry
InstElement	db	An element in an instantiation path
Instance	db	An instance proxy
Interpreter	tl	A generalization of script interpreters
KeyCode	lay	The namespace for the some key codes.
LEFDEFReaderConfiguration	db	Detailed LEF/DEF reader options
LayerInfo	db	A structure encapsulating the layer properties
LayerMap	db	An object representing an arbitrary mapping of physical layers to logical layers
LayerMapping	db	A layer mapping (source to target layout)
LayerProperties	lay	The layer properties structure
LayerPropertiesIterator	lay	Layer properties iterator
LayerPropertiesNode	lay	A layer properties node structure
LayerPropertiesNodeRef	lay	A class representing a reference to a layer properties node
Layout	db	The layout object
LayoutDiff	db	The layout compare tool
LayoutMetaInfo	db	A piece of layout meta information
LayoutQuery	db	A layout query
LayoutQueryIterator	db	Provides the results of the query
LayoutToNetlist	db	A generic framework for extracting netlists from layouts
LayoutToNetlist::BuildNetHierarchyMode	db	This class represents the LayoutToNetlist::BuildNetHierarchyMode enum
LayoutView	lay	The view object presenting one or more layout objects
LayoutView::SelectionMode	lay	Specifies how selected objects interact with already selected ones.
LayoutViewWidget	lay	
LayoutVsSchematic	db	A generic framework for doing LVS (layout vs. schematic)
Library	db	A Library
LoadLayoutOptions	db	Layout reader options

LoadLayoutOptions::CellConflictResolution	db	This enum specifies how cell conflicts are handled if a layout read into another layout and a cell name conflict arises.
LogEntryData	db	A generic log entry
Logger	tl	A logger
Macro	lay	A macro class
Macro::Format	lay	Specifies the format of a macro
Macro::Interpreter	lay	Specifies the interpreter used for executing a macro
MacroExecutionContext	lay	Support for various debugger features
MacroInterpreter	lay	A custom interpreter for a DSL (domain specific language)
MainWindow	lay	The main application window and central controller object
Manager	db	A transaction manager class
Marker	lay	The floating-point coordinate marker object
Matrix2d	db	A 2d matrix object used mainly for representing rotation and shear transformations.
Matrix3d	db	A 3d matrix object used mainly for representing rotation, shear, displacement and perspective transformations.
MessageBox	lay	Various methods to display message boxes
Metrics	db	This class represents the metrics type for \Region#width and related checks.
Net	db	A single net.
NetElement	db	A net element for the NetTracer net tracing facility
NetPinRef	db	A connection to an outgoing pin of the circuit.
NetSubcircuitPinRef	db	A connection to a pin of a subcircuit.
NetTerminalRef	db	A connection to a terminal of a device.
NetTracer	db	The net tracer feature
NetTracerConnectionInfo	db	Represents a single connection info line for the net tracer technology definition
NetTracerConnectivity	db	A connectivity description for the net tracer
NetTracerSymbolInfo	db	Represents a single symbol info line for the net tracer technology definition
NetTracerTechnologyComponent	db	Represents the technology information for the net tracer.
Netlist	db	The netlist top-level class

NetlistBrowserDialog	lay	Represents the netlist browser dialog.
NetlistCompareLogger	db	A base class for netlist comparer event receivers
NetlistComparer	db	Compares two netlists
NetlistCrossReference	db	Represents the identity mapping between the objects of two netlists.
NetlistCrossReference::CircuitPairData	db	A circuit match entry.
NetlistCrossReference::DevicePairData	db	A device match entry.
NetlistCrossReference::NetPairData	db	A net match entry.
NetlistCrossReference::NetPinRefPair	db	A match entry for a net pin pair.
NetlistCrossReference::NetSubcircuitPinRefPair	db	A match entry for a net subcircuit pin pair.
NetlistCrossReference::NetTerminalRefPair	db	A match entry for a net terminal pair.
NetlistCrossReference::PinPairData	db	A pin match entry.
NetlistCrossReference::Status	db	This class represents the NetlistCrossReference::Status enum
NetlistCrossReference::SubCircuitPairData	db	A subcircuit match entry.
NetlistDeviceExtractorLayerDefinition	db	Describes a layer used in the device extraction
NetlistObject	db	The base class for some netlist objects.
NetlistObjectPath	lay	An object describing the instantiation of a netlist object.
NetlistObjectsPath	lay	An object describing the instantiation of a single netlist object or a pair of those.
NetlistReader	db	Base class for netlist readers
NetlistSpiceReader	db	Implements a netlist Reader for the SPICE format.
NetlistSpiceReaderDelegate	db	Provides a delegate for the SPICE reader for translating device statements
NetlistSpiceWriter	db	Implements a netlist writer for the SPICE format.
NetlistSpiceWriterDelegate	db	Provides a delegate for the SPICE writer for doing special formatting for devices
NetlistWriter	db	Base class for netlist writers
ObjectInstPath	lay	A class describing a selected shape or instance
PCellDeclaration	db	A PCell declaration providing the parameters and code to produce the PCell
PCellDeclarationHelper	db	A helper class to simplify the declaration of a PCell (Python version)

PCellDeclarationHelper	db	A helper class to simplify the declaration of a PCell (Ruby version)
PCellParameterDeclaration	db	A PCell parameter declaration
PCellParameterState	db	Provides access to the attributes of a single parameter within \PCellParameterStates.
PCellParameterState::ParameterStateIcon	db	This enum specifies the icon shown next to the parameter in PCell parameter list.
PCellParameterStates	db	Provides access to the parameter states inside a 'callback' implementation of a PCell
ParentInstArray	db	A parent instance
ParseElementComponentsData	db	Supplies the return value for \NetlistSpiceReaderDelegate#parse_element_components.
ParseElementData	db	Supplies the return value for \NetlistSpiceReaderDelegate#parse_element.
Path	db	A path class
Pin	db	A pin of a circuit.
PixelBuffer	lay	A simplistic pixel buffer representing an image of ARGB32 or RGB32 values
Plugin	lay	The plugin object
PluginFactory	lay	The plugin framework's plugin factory object
Point	db	An integer point class
Polygon	db	A polygon class
PolygonFilter	db	A generic polygon filter adaptor
PolygonOperator	db	A generic polygon operator
PolygonToEdgeOperator	db	A generic polygon-to-edge operator
PolygonToEdgePairOperator	db	A generic polygon-to-edge-pair operator
PreferredOrientation	db	This class represents the PreferredOrientation enum used within polygon decomposition
Progress	tl	A progress reporter
PropertyConstraint	db	This class represents the property constraint for boolean and check functions.
RdbCategory	rdb	A category inside the report database
RdbCell	rdb	A cell inside the report database
RdbItem	rdb	An item inside the report database

RdbItemValue	rdb	A value object inside the report database
RdbReference	rdb	A cell reference inside the report database
Recipe	tl	A facility for providing reproducible recipes
RecursiveInstanceIterator	db	An iterator delivering instances recursively
RecursiveShapeIterator	db	An iterator delivering shapes recursively
Region	db	A region (a potentially complex area consisting of multiple polygons)
Region::OppositeFilter	db	This class represents the opposite error filter mode for \Region#separation and related checks.
Region::RectFilter	db	This class represents the error filter mode on rectangles for \Region#separation and related checks.
RelativeProgress	tl	A progress reporter counting progress in relative units
ReportDatabase	rdb	The report database object
SaveLayoutOptions	db	Options for saving layouts
Severity	db	This enum specifies the severity level for log entries.
Shape	db	An object representing a shape in the layout database
ShapeCollection	db	A base class for the shape collections (\Region, \Edges, \EdgePairs and \Texts)
ShapeProcessor	db	The shape processor (boolean, sizing, merge on shapes)
Shapes	db	A collection of shapes
SimplePolygon	db	A simple polygon class
SubCircuit	db	A subcircuit inside a circuit.
Technology	db	Represents a technology
TechnologyComponent	db	A part of a technology definition
Text	db	A text object
TextFilter	db	A generic text filter adaptor
TextGenerator	db	A text generator class
TextOperator	db	A generic text operator
TextToPolygonOperator	db	A generic text-to-polygon operator
Texts	db	Texts (a collection of texts)
TileOutputReceiver	db	A receiver abstraction for the tiling processor.

TilingProcessor	db	A processor for layout which distributes tasks over tiles
Timer	tl	A timer (stop watch)
Trans	db	A simple transformation
TrapezoidDecompositionMode	db	This class represents the TrapezoidDecompositionMode enum used within trapezoid decomposition
Utils	db	This namespace provides a collection of utility functions
VAlign	db	This class represents the vertical alignment modes.
VCplxTrans	db	A complex transformation
Value	tl	Encapsulates a value (preferably a plain data type) in an object
Vector	db	A integer vector class
ZeroDistanceMode	db	This class represents the zero_distance_mode type for \Region#width and related checks.

4.1. API reference - Class EmptyClass

Notation used in Ruby API documentation

Module: tl

Description:

Public constructors

new EmptyClass ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const EmptyClas other)	Assigns another object to self
[const]	new EmptyClass ptr	**dup**		Creates a copy of self

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const EmptyClass other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

	Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
dup	**Signature**: *[const]* new EmptyClass ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**Signature**: *[static]* new EmptyClass ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.

4.2. API reference - Class Value

Notation used in Ruby API documentation

Module: tl

Description: Encapsulates a value (preferably a plain data type) in an object

This class is provided to 'box' a value (encapsulate the value in an object). This class is required to interface to pointer or reference types in a method call. By using that class, the method can alter the value and thus implement 'out parameter' semantics. The value may be 'nil' which acts as a null pointer in pointer type arguments. This class has been introduced in version 0.22.

Public constructors

new Value ptr	**new**		Constructs a nil object.
new Value ptr	**new**	(variant value)	Constructs a non-nil object with the given value.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Value other)	Assigns another object to self
[const]	new Value ptr	**dup**		Creates a copy of self
[const]	string	**to_s**		Convert this object to a string
[const]	variant	**value**		Gets the actual value.
	void	**value=**	(variant value)	Set the actual value.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead

[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Value other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Value ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

(1) Signature: *[static]* new Value ptr **new**

Description: Constructs a nil object.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Value ptr **new** (variant value)

Description: Constructs a non-nil object with the given value.

This constructor has been introduced in version 0.22.

Python specific notes:
This method is the default initializer of the object.

to_s

Signature: *[const]* string **to_s**

Description: Convert this object to a string

Python specific notes:
This method is also available as 'str(object)'.

value	**Signature**: *[const]* variant **value** **Description**: Gets the actual value. **Python specific notes:** The object exposes a readable attribute 'value'. This is the getter.
value=	**Signature**: void **value=** (variant value) **Description**: Set the actual value. **Python specific notes:** The object exposes a writable attribute 'value'. This is the setter.

4.3. API reference - Class Interpreter

Notation used in Ruby API documentation

Module: tl

Description: A generalization of script interpreters

The main purpose of this class is to provide cross-language call options. Using the Python interpreter, it is possible to execute Python code from Ruby for example.

The following example shows how to use the interpreter class to execute Python code from Ruby and how to pass values from Ruby to Python and back using the Value wrapper object:

```
pya = RBA::Interpreter::python_interpreter
out_param = RBA::Value::new(17)
pya.define_variable("out_param", out_param)
pya.eval_string(<<END)
print("This is Python now!")
out_param.value = out_param.value + 25
END
puts out_param.value  # gives '42'
```

This class was introduced in version 0.27.5.

Public constructors

new Interpreter ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**define_variable**	(string name, variant value)	Defines a (global) variable with the given name and value
	variant	**eval_expr**	(string string, string filename = nil, int line = 1)	Executes the expression inside the given string and returns the result value
	void	**eval_string**	(string string, string filename = nil, int line = 1)	Executes the code inside the given string

void	load_file	(string path)	Loads the given file into the interpreter

Public static methods and constants

Interpreter ptr	python_interpreter	Gets the instance of the Python interpreter
Interpreter ptr	ruby_interpreter	Gets the instance of the Ruby interpreter

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void _**manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is

	known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
define_variable	**Signature**: void **define_variable** (string name, variant value) **Description**: Defines a (global) variable with the given name and value You can use the Value class to provide 'out' or 'inout' parameters which can be modified by code executed inside the interpreter and read back by the caller.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
eval_expr	**Signature**: variant **eval_expr** (string string, string filename = nil, int line = 1) **Description**: Executes the expression inside the given string and returns the result value Use 'filename' and 'line' to indicate the original source for the error messages.
eval_string	**Signature**: void **eval_string** (string string, string filename = nil, int line = 1) **Description**: Executes the code inside the given string Use 'filename' and 'line' to indicate the original source for the error messages.

is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
load_file	**Signature**: void **load_file** (string path) **Description**: Loads the given file into the interpreter This will execute the code inside the file.
new	**Signature**: *[static]* new Interpreter ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
python_interpreter	**Signature**: *[static]* Interpreter ptr **python_interpreter** **Description**: Gets the instance of the Python interpreter
ruby_interpreter	**Signature**: *[static]* Interpreter ptr **ruby_interpreter** **Description**: Gets the instance of the Ruby interpreter

4.4. API reference - Class Logger

Notation used in Ruby API documentation

Module: tl

Description: A logger

The logger outputs messages to the log channels. If the log viewer is open, the log messages will be shown in the logger view. Otherwise they will be printed to the terminal on Linux for example.

A code example:

```
RBA::Logger::error("An error message")
RBA::Logger::warn("A warning")
```

This class has been introduced in version 0.23.

Public constructors

new Logger ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Logger other)	Assigns another object to self
[const]	new Logger ptr	**dup**		Creates a copy of self

Public static methods and constants

	void	**error**	(string msg)	Writes the given string to the error channel
	void	**info**	(string msg)	Writes the given string to the info channel
	void	**log**	(string msg)	Writes the given string to the log channel
	int	**verbosity**		Returns the verbosity level

	void	verbosity=	(int v)	Sets the verbosity level for the application
	void	warn	(string msg)	Writes the given string to the warning channel

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void _**manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Logger other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Logger ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

error

Signature: *[static]* void **error** (string msg)

Description: Writes the given string to the error channel

The error channel is formatted as an error (i.e. red in the logger window) and output unconditionally.

info

Signature: *[static]* void **info** (string msg)

Description: Writes the given string to the info channel

The info channel is printed as neutral messages unconditionally.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

log

Signature: *[static]* void **log** (string msg)

Description: Writes the given string to the log channel

Log messages are printed as neutral messages and are output only if the verbosity is above 0.

new

Signature: *[static]* new Logger ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

verbosity

Signature: *[static]* int **verbosity**

Description: Returns the verbosity level

The verbosity level is defined by the application (see -d command line option for example). Level 0 is silent, levels 10, 20, 30 etc. denote levels with increasing verbosity. 11, 21, 31 .. are sublevels which also enable timing logs in addition to messages.

Python specific notes:
The object exposes a readable attribute 'verbosity'. This is the getter.

verbosity=

Signature: *[static]* void **verbosity=** (int v)

Description: Sets the verbosity level for the application

See verbosity for a definition of the verbosity levels. Please note that this method changes the verbosity level for the whole application.

Python specific notes:
The object exposes a writable attribute 'verbosity'. This is the setter.

warn

Signature: *[static]* void **warn** (string msg)

Description: Writes the given string to the warning channel

The warning channel is formatted as a warning (i.e. blue in the logger window) and output unconditionally.

4.5. API reference - Class Timer

Notation used in Ruby API documentation

Module: tl

Description: A timer (stop watch)

The timer provides a way to measure CPU time. It provides two basic methods: start and stop. After it has been started and stopped again, the time can be retrieved using the user and sys attributes, i.e.:

```
t = RBA::Timer::new
t.start
# ... do something
t.stop
puts "it took #{t.sys} seconds (kernel), #{t.user} seconds (user) on the CPU"
```

The time is reported in seconds.

This class has been introduced in version 0.23.

Public constructors

new Timer ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Timer other)	Assigns another object to self
[const]	new Timer ptr	**dup**		Creates a copy of self
	void	**start**		Starts the timer
	void	**stop**		Stops the timer
[const]	double	**sys**		Returns the elapsed CPU time in kernel mode from start to stop in seconds
[const]	string	**to_s**		Produces a string with the currently elapsed times

[const]	double	**user**	Returns the elapsed CPU time in user mode from start to stop in seconds
[const]	double	**wall**	Returns the elapsed real time from start to stop in seconds

Public static methods and constants

unsigned long	**memory_size**	Gets the current memory usage of the process in Bytes

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is

	known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
assign	**Signature**: void **assign** (const Timer other) **Description**: Assigns another object to self
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
dup	**Signature**: *[const]* new Timer ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.

memory_size	**Signature**: *[static]* unsigned long **memory_size** **Description**: Gets the current memory usage of the process in Bytes This method has been introduced in version 0.27.
new	**Signature**: *[static]* new Timer ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
start	**Signature**: void **start** **Description**: Starts the timer
stop	**Signature**: void **stop** **Description**: Stops the timer
sys	**Signature**: *[const]* double **sys** **Description**: Returns the elapsed CPU time in kernel mode from start to stop in seconds
to_s	**Signature**: *[const]* string **to_s** **Description**: Produces a string with the currently elapsed times **Python specific notes:** This method is also available as 'str(object)'.
user	**Signature**: *[const]* double **user** **Description**: Returns the elapsed CPU time in user mode from start to stop in seconds
wall	**Signature**: *[const]* double **wall** **Description**: Returns the elapsed real time from start to stop in seconds This method has been introduced in version 0.26.

4.6. API reference - Class Progress

Notation used in Ruby API documentation

Module: tl

Description: A progress reporter

This is the base class for all progress reporter objects. Progress reporter objects are used to report the progress of some operation and to allow aborting an operation. Progress reporter objects must be triggered periodically, i.e. a value must be set. On the display side, a progress bar usually is used to represent the progress of an operation.

Actual implementations of the progress reporter class are RelativeProgress and AbsoluteProgress.

This class has been introduced in version 0.23.

Public constructors

new Progress ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	string	**desc**		Gets the description text of the progress object
	void	**desc=**	(string desc)	Sets the description text of the progress object
	void	**title=**	(string title)	Sets the title text of the progress object

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

	Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
desc	**Signature**: *[const]* string **desc** **Description**: Gets the description text of the progress object **Python specific notes:** The object exposes a readable attribute 'desc'. This is the getter.
desc=	**Signature**: void **desc=** (string desc) **Description**: Sets the description text of the progress object **Python specific notes:** The object exposes a writable attribute 'desc'. This is the setter.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**Signature**: *[static]* new Progress ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
title=	**Signature**: void **title=** (string title) **Description**: Sets the title text of the progress object Initially the title is equal to the description. **Python specific notes:** The object exposes a writable attribute 'title'. This is the setter.

4.7. API reference - Class AbstractProgress

Notation used in Ruby API documentation

Module: tl

Description: The abstract progress reporter

Class hierarchy: AbstractProgress » Progress

The abstract progress reporter acts as a 'bracket' for a sequence of operations which are connected logically. For example, a DRC script consists of multiple operations. An abstract progress reportert is instantiated during the run time of the DRC script. This way, the application leaves the UI open while the DRC executes and log messages can be collected.

The abstract progress does not have a value.

This class has been introduced in version 0.27.

Public constructors

new AbstractProgress ptr	**new**	(string desc)		Creates an abstract progress reporter with the given description

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const AbstractPrc other)	Assigns another object to self
[const]	new AbstractProgress ptr	**dup**		Creates a copy of self

Detailed description

_create

Signature: void _create

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _destroy

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const AbstractProgress other)

Description: Assigns another object to self

dup

Signature: *[const]* new AbstractProgress ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

new

Signature: *[static]* new AbstractProgress ptr **new** (string desc)

Description: Creates an abstract progress reporter with the given description

Python specific notes:
This method is the default initializer of the object.

4.8. API reference - Class RelativeProgress

Notation used in Ruby API documentation

Module: tl

Description: A progress reporter counting progress in relative units

Class hierarchy: RelativeProgress » Progress

A relative progress reporter counts from 0 to some maximum value representing 0 to 100 percent completion of a task. The progress can be configured to have a description text, a title and a format. The "inc" method increments the value, the "set" or "value=" methods set the value to a specific value.

While one of these three methods is called, they will run the event loop in regular intervals. That makes the application respond to mouse clicks, specifically the Cancel button on the progress bar. If that button is clicked, an exception will be raised by these methods.

The progress object must be destroyed explicitly in order to remove the progress status bar.

A code example:

```
p = RBA::RelativeProgress::new("test", 10000000)
begin
  10000000.times { p.inc }
ensure
  p.destroy
end
```

This class has been introduced in version 0.23.

Public constructors

new RelativeProgress ptr	**new**	(string desc, unsigned long max_value)	Creates a relative progress reporter with the given description and maximum value
new RelativeProgress ptr	**new**	(string desc, unsigned long max_value, unsigned long yield_interval)	Creates a relative progress reporter with the given description and maximum value

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const RelativeProgress other)	Assigns another object to self

[const]	new RelativeProgress ptr	**dup**		Creates a copy of self
	void	**format=**	(string format)	sets the output format (sprintf notation) for the progress text
	RelativeProgress	**inc**		Increments the progress value
	void	**set**	(unsigned long value, bool force_yield)	Sets the progress value
	void	**value=**	(unsigned long value)	Sets the progress value

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const RelativeProgress other)

Description: Assigns another object to self

dup

Signature: *[const]* new RelativeProgress ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

format=

Signature: void **format=** (string format)

Description: sets the output format (sprintf notation) for the progress text

Python specific notes:
The object exposes a writable attribute 'format'. This is the setter.

inc

Signature: RelativeProgress **inc**

Description: Increments the progress value

new

(1) Signature: *[static]* new RelativeProgress ptr **new** (string desc, unsigned long max_value)

Description: Creates a relative progress reporter with the given description and maximum value

The reported progress will be 0 to 100% for values between 0 and the maximum value. The values are always integers. Double values cannot be used property.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new RelativeProgress ptr **new** (string desc, unsigned long max_value, unsigned long yield_interval)

Description: Creates a relative progress reporter with the given description and maximum value

The reported progress will be 0 to 100% for values between 0 and the maximum value. The values are always integers. Double values cannot be used property.

The yield interval specifies, how often the event loop will be triggered. When the yield interval is 10 for example, the event loop will be executed every tenth call of inc or set.

Python specific notes:
This method is the default initializer of the object.

set

Signature: void **set** (unsigned long value, bool force_yield)

Description: Sets the progress value

This method is equivalent to value=, but it allows forcing the event loop to be triggered. If "force_yield" is true, the event loop will be triggered always, irregardless of the yield interval specified in the constructor.

value=

Signature: void **value=** (unsigned long value)

Description: Sets the progress value

Python specific notes:
The object exposes a writable attribute 'value'. This is the setter.

4.9. API reference - Class AbsoluteProgress

Notation used in Ruby API documentation

Module: tl

Description: A progress reporter counting progress in absolute units

Class hierarchy: AbsoluteProgress » Progress

An absolute progress reporter counts from 0 upwards without a known limit. A unit value is used to convert the value to a bar value. One unit corresponds to 1% on the bar. For formatted output, a format string can be specified as well as a unit value by which the current value is divided before it is formatted.

The progress can be configured to have a description text, a title and a format. The "inc" method increments the value, the "set" or "value=" methods set the value to a specific value.

While one of these three methods is called, they will run the event loop in regular intervals. That makes the application respond to mouse clicks, specifically the Cancel button on the progress bar. If that button is clicked, an exception will be raised by these methods.

The progress object must be destroyed explicitly in order to remove the progress status bar.

The following sample code creates a progress bar which displays the current count as "Megabytes". For the progress bar, one percent corresponds to 16 kByte:

```
p = RBA::AbsoluteProgress::new("test")
p.format = "%.2f MBytes"
p.unit = 1024*16
p.format_unit = 1024*1024
begin
  10000000.times { p.inc }
ensure
  p.destroy
end
```

This class has been introduced in version 0.23.

Public constructors

new AbsoluteProgress ptr	**new**	(string desc)	Creates an absolute progress reporter with the given description
new AbsoluteProgress ptr	**new**	(string desc, unsigned long yield_interval)	Creates an absolute progress reporter with the given description

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.

	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const AbsoluteProgress other)	Assigns another object to self
[const]	new AbsoluteProgress ptr	**dup**		Creates a copy of self
	void	**format=**	(string format)	sets the output format (sprintf notation) for the progress text
	void	**format_unit=**	(double unit)	Sets the format unit
	AbsoluteProgress	**inc**		Increments the progress value
	void	**set**	(unsigned long value, bool force_yield)	Sets the progress value
	void	**unit=**	(double unit)	Sets the unit
	void	**value=**	(unsigned long value)	Sets the progress value

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

	After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
assign	**Signature**: void **assign** (const AbsoluteProgress other) **Description**: Assigns another object to self
dup	**Signature**: *[const]* new AbsoluteProgress ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
format=	**Signature**: void **format=** (string format) **Description**: sets the output format (sprintf notation) for the progress text **Python specific notes:** The object exposes a writable attribute 'format'. This is the setter.
format_unit=	**Signature**: void **format_unit=** (double unit) **Description**: Sets the format unit This is the unit used for formatted output. The current count is divided by the format unit to render the value passed to the format string. **Python specific notes:** The object exposes a writable attribute 'format_unit'. This is the setter.
inc	**Signature**: AbsoluteProgress **inc** **Description**: Increments the progress value
new	**(1) Signature**: *[static]* new AbsoluteProgress ptr **new** (string desc) **Description**: Creates an absolute progress reporter with the given description **Python specific notes:** This method is the default initializer of the object. **(2) Signature**: *[static]* new AbsoluteProgress ptr **new** (string desc, unsigned long yield_interval) **Description**: Creates an absolute progress reporter with the given description

The yield interval specifies, how often the event loop will be triggered. When the yield interval is 10 for example, the event loop will be executed every tenth call of inc or set.

Python specific notes:
This method is the default initializer of the object.

set

Signature: void **set** (unsigned long value, bool force_yield)

Description: Sets the progress value

This method is equivalent to value=, but it allows forcing the event loop to be triggered. If "force_yield" is true, the event loop will be triggered always, irregardless of the yield interval specified in the constructor.

unit=

Signature: void **unit=** (double unit)

Description: Sets the unit

Specifies the count value corresponding to 1 percent on the progress bar. By default, the current value divided by the unit is used to create the formatted value from the output string. Another attribute is provided (format_unit=) to specify a separate unit for that purpose.

Python specific notes:
The object exposes a writable attribute 'unit'. This is the setter.

value=

Signature: void **value=** (unsigned long value)

Description: Sets the progress value

Python specific notes:
The object exposes a writable attribute 'value'. This is the setter.

4.10. API reference - Class ExpressionContext

Notation used in Ruby API documentation

Module: tl

Description: Represents the context of an expression evaluation

The context provides a variable namespace for the expression evaluation.

This class has been introduced in version 0.26 when Expression was separated into the execution and context part.

Public constructors

new ExpressionContext ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const ExpressionCont other)	Assigns another object to self
[const]	new ExpressionContext ptr	**dup**		Creates a copy of self
	variant	**eval**	(string expr)	Compiles and evaluates the given expression in this context
	void	**var**	(string name, variant value)	Defines a variable with the given name and value

Public static methods and constants

void	**global_var**	(string name, variant value)	Defines a global variable with the given name and value

Deprecated methods (protected, public, static, non-static and constructors)

void	**create**		Use of this method is deprecated. Use _create instead

	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const ExpressionContext other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new ExpressionContext ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

eval

Signature: variant **eval** (string expr)

Description: Compiles and evaluates the given expression in this context

This method has been introduced in version 0.26.

global_var

Signature: *[static]* void **global_var** (string name, variant value)

Description: Defines a global variable with the given name and value

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new ExpressionContext ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

var

Signature: void **var** (string name, variant value)

Description: Defines a variable with the given name and value

4.11. API reference - Class Expression

Notation used in Ruby API documentation

Module: tl

Description: Evaluation of Expressions

Class hierarchy: Expression » ExpressionContext

This class allows evaluation of expressions. Expressions are used in many places throughout KLayout and provide computation features for various applications. Having a script language, there is no real use for expressions inside a script client. This class is provided mainly for testing purposes.

An expression is 'compiled' into an Expression object and can be evaluated multiple times.

This class has been introduced in version 0.25. In version 0.26 it was separated into execution and context.

Public constructors

new Expression ptr	**new**	(string expr)	Creates an expression evaluator
new Expression ptr	**new**	(string expr, map<string,variant> variables)	Creates an expression evaluator

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	variant	**eval**		Evaluates the current expression and returns the result
	void	**text=**	(string expr)	Sets the given text as the expression.

Public static methods and constants

variant	**eval**	(string expr)	A convience function to evaluate the given expression and directly return the result

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

eval

(1) Signature: variant **eval**

Description: Evaluates the current expression and returns the result

Python specific notes:
This instance method is available as '_inst_eval' in Python.

(2) Signature: *[static]* variant **eval** (string expr)

Description: A convience function to evaluate the given expression and directly return the result

This is a static method that does not require instantiation of the expression object first.

Python specific notes:
This class method is available as '_class_eval' in Python.

new	**(1) Signature**: *[static]* new Expression ptr **new** (string expr) **Description**: Creates an expression evaluator **Python specific notes:** This method is the default initializer of the object. **(2) Signature**: *[static]* new Expression ptr **new** (string expr, map<string,variant> variables) **Description**: Creates an expression evaluator This version of the constructor takes a hash of variables available to the expressions. **Python specific notes:** This method is the default initializer of the object.
text=	**Signature**: void **text=** (string expr) **Description**: Sets the given text as the expression. **Python specific notes:** The object exposes a writable attribute 'text'. This is the setter.

4.12. API reference - Class GlobPattern

Notation used in Ruby API documentation

Module: tl

Description: A glob pattern matcher

This class is provided to make KLayout's glob pattern matching available to scripts too. The intention is to provide an implementation which is compatible with KLayout's pattern syntax.

This class has been introduced in version 0.26.

Public constructors

new GlobPattern ptr	**new**	(string pattern)		Creates a new glob pattern match object

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const GlobPattern other)	Assigns another object to self
[const]	bool	**case_sensitive**		Gets a value indicating whether the glob pattern match is case sensitive.
	void	**case_sensitive=**	(bool case_sensitive)	Sets a value indicating whether the glob pattern match is case sensitive.
[const]	new GlobPattern ptr	**dup**		Creates a copy of self
[const]	bool	**head_match**		Gets a value indicating whether trailing characters are allowed.
	void	**head_match=**	(bool head_match)	Sets a value indicating whether trailing characters are allowed.
[const]	variant	**match**	(string subject)	Matches the subject string against the pattern.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const GlobPattern other)

Description: Assigns another object to self

case_sensitive

Signature: *[const]* bool **case_sensitive**

Description: Gets a value indicating whether the glob pattern match is case sensitive.

Python specific notes:
The object exposes a readable attribute 'case_sensitive'. This is the getter.

case_sensitive=

Signature: void **case_sensitive=** (bool case_sensitive)

Description: Sets a value indicating whether the glob pattern match is case sensitive.

Python specific notes:
The object exposes a writable attribute 'case_sensitive'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new GlobPattern ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

head_match

Signature: *[const]* bool **head_match**

Description: Gets a value indicating whether trailing characters are allowed.

Python specific notes:

	The object exposes a readable attribute 'head_match'. This is the getter.
head_match=	**Signature**: void **head_match=** (bool head_match) **Description**: Sets a value indicating whether trailing characters are allowed. If this predicate is false, the glob pattern needs to match the full subject string. If true, the match function will ignore trailing characters and return true if the front part of the subject string matches. **Python specific notes:** The object exposes a writable attribute 'head_match'. This is the setter.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
match	**Signature**: *[const]* variant **match** (string subject) **Description**: Matches the subject string against the pattern. Returns nil if the subject string does not match the pattern. Otherwise returns a list with the substrings captured in round brackets.
new	**Signature**: *[static]* new GlobPattern ptr **new** (string pattern) **Description**: Creates a new glob pattern match object **Python specific notes:** This method is the default initializer of the object.

4.13. API reference - Class Executable

Notation used in Ruby API documentation

Module: tl

Description: A generic executable object

Class hierarchy: Executable

This object is a delegate for implementing the actual function of some generic executable function. In addition to the plain execution, if offers a post-mortem cleanup callback which is always executed, even if execute's implementation is cancelled in the debugger.

Parameters are kept as a generic key/value map.

This class has been introduced in version 0.27.

Public constructors

new Executable ptr	**new**	Creates a new object of this class

Public methods

	void	**_assign**	(const Executable other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new Executable ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Executable other)	Assigns another object to self
[virtual]	void	**cleanup**		Reimplement this method to provide post-mortem cleanup functionality.
[const]	new Executable ptr	**dup**		Creates a copy of self
[virtual]	variant	**execute**		Reimplement this method to provide the functionality of the executable.

Deprecated methods (protected, public, static, non-static and constructors)

void	**create**	Use of this method is deprecated. Use _create instead

	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_assign

Signature: void **_assign** (const Executable other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new Executable ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Executable other)

Description: Assigns another object to self

cleanup

Signature: *[virtual]* void **cleanup**

Description: Reimplement this method to provide post-mortem cleanup functionality.

This method is always called after execute terminated.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Executable ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

execute

Signature: *[virtual]* variant **execute**

Description: Reimplement this method to provide the functionality of the executable.

This method is supposed to execute the operation with the given parameters and return the desired output.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new Executable ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

4.14. API reference - Class Recipe

Notation used in Ruby API documentation

Module: tl

Description: A facility for providing reproducible recipes

The idea of this facility is to provide a service by which an object can be reproduced in a parametrized way. The intended use case is a DRC report for example, where the DRC script is the generator.

In this use case, the DRC engine will register a recipe. It will put the serialized version of the recipe into the DRC report. If the user requests a re-run of the DRC, the recipe will be called and the implementation is supposed to deliver a new database.

To register a recipe, reimplement the Recipe class and create an instance. To serialize a recipe, use "generator", to execute the recipe, use "make".

Parameters are kept as a generic key/value map.

This class has been introduced in version 0.26.

Public constructors

new Recipe ptr	**new**	(string name, string description = "")	Creates a new recipe object with the given name and (optional) description

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	string	**description**		Gets the description of the recipe.
[virtual,const]	Executable ptr	**executable**	(map<string,variant> params)	Reimplement this method to provide an executable object for the actual implementation.
	string	**generator**	(map<string,v params)	Delivers the generator string from the given parameters.
[const]	string	**name**		Gets the name of the recipe.

Public static methods and constants

variant	**make**	(string generator, map<string,variant> add_params = {})	Executes the recipe given by the generator string.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void _**manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void _**unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

description

Signature: *[const]* string **description**

Description: Gets the description of the recipe.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

executable

Signature: *[virtual,const]* Executable ptr **executable** (map<string,variant> params)

Description: Reimplement this method to provide an executable object for the actual implementation.

The reasoning behind this architecture is to supply a cleanup callback. This is useful when the actual function is executed as a script and the script terminates in the debugger. The cleanup callback allows implementing any kind of post-mortem action despite being cancelled in the debugger.

This method has been introduced in version 0.27 and replaces 'execute'.

generator

Signature: string **generator** (map<string,variant> params)

Description: Delivers the generator string from the given parameters.

The generator string can be used with make to re-run the recipe.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

make

Signature: *[static]* variant **make** (string generator, map<string,variant> add_params = {})

Description: Executes the recipe given by the generator string.

The generator string is the one delivered with generator. Additional parameters can be passed in "add_params". They have lower priority than the parameters kept inside the generator string.

name

Signature: *[const]* string **name**

Description: Gets the name of the recipe.

new

Signature: *[static]* new Recipe ptr **new** (string name, string description = "")

Description: Creates a new recipe object with the given name and (optional) description

Python specific notes:
This method is the default initializer of the object.

4.15. API reference - Class PixelBuffer

Notation used in Ruby API documentation

Module: lay

Description: A simplistic pixel buffer representing an image of ARGB32 or RGB32 values

This object is mainly provided for offline rendering of layouts in Qt-less environments. It supports a rectangular pixel space with color values encoded in 32bit integers. It supports transparency through an optional alpha channel. The color format for a pixel is "0xAARRGGBB" where 'AA' is the alpha value which is ignored in non-transparent mode.

This class supports basic operations such as initialization, single-pixel access and I/O to PNG.

This class has been introduced in version 0.28.

Public constructors

new PixelBuffer ptr	**new**	(unsigned int width, unsigned int height)	Creates a pixel buffer object

Public methods

[const]	bool	**!=**	(const PixelBuffer other)	Returns a value indicating whether self is not identical to the other image
[const]	bool	**==**	(const PixelBuffer other)	Returns a value indicating whether self is identical to the other image
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const PixelBuffer other)	Assigns another object to self
[const]	PixelBuffer	**diff**	(const PixelBuffer other)	Creates a difference image
[const]	new PixelBuffer ptr	**dup**		Creates a copy of self
	void	**fill**	(unsigned int color)	Fills the pixel buffer with the given pixel value
	void	**fill**	(QColor color)	Fills the pixel buffer with the given QColor
[const]	unsigned int	**height**		Gets the height of the pixel buffer in pixels

	void	**patch**	(const PixelBuffer other)	Patches another pixel buffer into this one
[const]	unsigned int	**pixel**	(unsigned int x, unsigned int y)	Gets the value of the pixel at position x, y
	void	**set_pixel**	(unsigned int x, unsigned int y, unsigned int c)	Sets the value of the pixel at position x, y
	void	**swap**	(PixelBuffer other)	Swaps data with another PixelBuffer object
[const]	bytes	**to_png_data**		Converts the pixel buffer to a PNG byte stream
[const]	QImage	**to_qimage**		Converts the pixel buffer to a QImage object
[const]	bool	**transparent**		Gets a flag indicating whether the pixel buffer supports an alpha channel
	void	**transparent=**	(bool t)	Sets a flag indicating whether the pixel buffer supports an alpha channel
[const]	unsigned int	**width**		Gets the width of the pixel buffer in pixels
[const]	void	**write_png**	(string file)	Writes the pixel buffer to a PNG file

Public static methods and constants

PixelBuffer	**from_png_data**	(bytes data)	Reads the pixel buffer from a PNG byte stream
PixelBuffer	**from_qimage**	(const QImage qimage)	Creates a pixel buffer object from a QImage object
PixelBuffer	**read_png**	(string file)	Reads the pixel buffer from a PNG file

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const PixelBuffer other)

Description: Returns a value indicating whether self is not identical to the other image

==	**Signature**: *[const]* bool **==** (const PixelBuffer other) **Description**: Returns a value indicating whether self is identical to the other image
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
assign	**Signature**: void **assign** (const PixelBuffer other) **Description**: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

diff

Signature: *[const]* PixelBuffer **diff** (const PixelBuffer other)

Description: Creates a difference image

This method is provided to support transfer of image differences - i.e. small updates instead of full images. It works for non-transparent images only and generates an image with transpareny enabled and with the new pixel values for pixels that have changed. The alpha value will be 0 for identical images and 255 for pixels with different values. This way, the difference image can be painted over the original image to generate the new image.

dup

Signature: *[const]* new PixelBuffer ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

fill

(1) Signature: void **fill** (unsigned int color)

Description: Fills the pixel buffer with the given pixel value

(2) Signature: void **fill** (QColor color)

Description: Fills the pixel buffer with the given QColor

from_png_data

Signature: *[static]* PixelBuffer **from_png_data** (bytes data)

Description: Reads the pixel buffer from a PNG byte stream

This method may not be available if PNG support is not compiled into KLayout.

from_qimage

Signature: *[static]* PixelBuffer **from_qimage** (const QImage qimage)

Description: Creates a pixel buffer object from a QImage object

height

Signature: *[const]* unsigned int **height**

Description: Gets the height of the pixel buffer in pixels

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new PixelBuffer ptr **new** (unsigned int width, unsigned int height)

Description: Creates a pixel buffer object

width:	The width in pixels
height:	The height in pixels

The pixels are basically uninitialized. You will need to use fill to initialize them to a certain value.

Python specific notes:
This method is the default initializer of the object.

patch

Signature: void **patch** (const PixelBuffer other)

Description: Patches another pixel buffer into this one

This method is the inverse of diff - it will patch the difference image created by diff into this pixel buffer. Note that this method will not do true alpha blending and requires the other pixel buffer to have the same format than self. Self will be modified by this operation.

pixel

Signature: *[const]* unsigned int **pixel** (unsigned int x, unsigned int y)

Description: Gets the value of the pixel at position x, y

read_png

Signature: *[static]* PixelBuffer **read_png** (string file)

Description: Reads the pixel buffer from a PNG file

This method may not be available if PNG support is not compiled into KLayout.

set_pixel

Signature: void **set_pixel** (unsigned int x, unsigned int y, unsigned int c)

Description: Sets the value of the pixel at position x, y

swap

Signature: void **swap** (PixelBuffer other)

Description: Swaps data with another PixelBuffer object

to_png_data

Signature: *[const]* bytes **to_png_data**

Description: Converts the pixel buffer to a PNG byte stream

This method may not be available if PNG support is not compiled into KLayout.

to_qimage

Signature: *[const]* QImage **to_qimage**

Description: Converts the pixel buffer to a QImage object

transparent	**Signature**: *[const]* bool **transparent** **Description**: Gets a flag indicating whether the pixel buffer supports an alpha channel **Python specific notes:** The object exposes a readable attribute 'transparent'. This is the getter.
transparent=	**Signature**: void **transparent=** (bool t) **Description**: Sets a flag indicating whether the pixel buffer supports an alpha channel By default, the pixel buffer does not support an alpha channel. **Python specific notes:** The object exposes a writable attribute 'transparent'. This is the setter.
width	**Signature**: *[const]* unsigned int **width** **Description**: Gets the width of the pixel buffer in pixels
write_png	**Signature**: *[const]* void **write_png** (string file) **Description**: Writes the pixel buffer to a PNG file This method may not be available if PNG support is not compiled into KLayout.

4.16. API reference - Class BitmapBuffer

Notation used in Ruby API documentation

Module: lay

Description: A simplistic pixel buffer representing monochrome image

This object is mainly provided for offline rendering of layouts in Qt-less environments. It supports a rectangular pixel space with color values encoded in single bits.

This class supports basic operations such as initialization, single-pixel access and I/O to PNG.

This class has been introduced in version 0.28.

Public constructors

new BitmapBuffer ptr	**new**	(unsigned int width, unsigned int height)	Creates a pixel buffer object

Public methods

[const]	bool	**!=**	(const BitmapBuffer other)	Returns a value indicating whether self is not identical to the other image
[const]	bool	**==**	(const BitmapBuffer other)	Returns a value indicating whether self is identical to the other image
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const BitmapBuffer other)	Assigns another object to self
[const]	new BitmapBuffer ptr	**dup**		Creates a copy of self
	void	**fill**	(bool color)	Fills the pixel buffer with the given pixel value
[const]	unsigned int	**height**		Gets the height of the pixel buffer in pixels
[const]	bool	**pixel**	(unsigned int x, unsigned int y)	Gets the value of the pixel at position x, y

	void	**set_pixel**	(unsigned int x, unsigned int y, bool c)	Sets the value of the pixel at position x, y
	void	**swap**	(BitmapBuffer other)	Swaps data with another BitmapBuffer object
[const]	bytes	**to_png_data**		Converts the pixel buffer to a PNG byte stream
[const]	QImage	**to_qimage**		Converts the pixel buffer to a QImage object
[const]	unsigned int	**width**		Gets the width of the pixel buffer in pixels
[const]	void	**write_png**	(string file)	Writes the pixel buffer to a PNG file

Public static methods and constants

	BitmapBuffer	**from_png_data**	(bytes data)	Reads the pixel buffer from a PNG byte stream
	BitmapBuffer	**from_qimage**	(const QImage qimage)	Creates a pixel buffer object from a QImage object
	BitmapBuffer	**read_png**	(string file)	Reads the pixel buffer from a PNG file

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const BitmapBuffer other)

Description: Returns a value indicating whether self is not identical to the other image

==

Signature: *[const]* bool **==** (const BitmapBuffer other)

Description: Returns a value indicating whether self is identical to the other image

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const BitmapBuffer other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

	Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
dup	**Signature**: *[const]* new BitmapBuffer ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
fill	**Signature**: void **fill** (bool color) **Description**: Fills the pixel buffer with the given pixel value
from_png_data	**Signature**: *[static]* BitmapBuffer **from_png_data** (bytes data) **Description**: Reads the pixel buffer from a PNG byte stream This method may not be available if PNG support is not compiled into KLayout.
from_qimage	**Signature**: *[static]* BitmapBuffer **from_qimage** (const QImage qimage) **Description**: Creates a pixel buffer object from a QImage object
height	**Signature**: *[const]* unsigned int **height** **Description**: Gets the height of the pixel buffer in pixels
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**Signature**: *[static]* new BitmapBuffer ptr **new** (unsigned int width, unsigned int height) **Description**: Creates a pixel buffer object **width**: The width in pixels **height**: The height in pixels The pixels are basically uninitialized. You will need to use fill to initialize them to a certain value. **Python specific notes:** This method is the default initializer of the object.
pixel	**Signature**: *[const]* bool **pixel** (unsigned int x, unsigned int y) **Description**: Gets the value of the pixel at position x, y

read_png	**Signature**: *[static]* BitmapBuffer **read_png** (string file) **Description**: Reads the pixel buffer from a PNG file This method may not be available if PNG support is not compiled into KLayout.
set_pixel	**Signature**: void **set_pixel** (unsigned int x, unsigned int y, bool c) **Description**: Sets the value of the pixel at position x, y
swap	**Signature**: void **swap** (BitmapBuffer other) **Description**: Swaps data with another BitmapBuffer object
to_png_data	**Signature**: *[const]* bytes **to_png_data** **Description**: Converts the pixel buffer to a PNG byte stream This method may not be available if PNG support is not compiled into KLayout.
to_qimage	**Signature**: *[const]* QImage **to_qimage** **Description**: Converts the pixel buffer to a QImage object
width	**Signature**: *[const]* unsigned int **width** **Description**: Gets the width of the pixel buffer in pixels
write_png	**Signature**: *[const]* void **write_png** (string file) **Description**: Writes the pixel buffer to a PNG file This method may not be available if PNG support is not compiled into KLayout.

4.17. API reference - Class Box

Notation used in Ruby API documentation

Module: db

Description: A box class with integer coordinates

This object represents a box (a rectangular shape).

The definition of the attributes is: p1 is the lower left point, p2 the upper right one. If a box is constructed from two points (or four coordinates), the coordinates are sorted accordingly.

A box can be empty. An empty box represents no area (not even a point). Empty boxes behave neutral with respect to most operations. Empty boxes return true on empty?.

A box can be a point or a single line. In this case, the area is zero but the box still can overlap other boxes for example and it is not empty.

See The Database API for more details about the database objects.

Public constructors

new Box ptr	**new**	(const DBox dbox)	Creates an integer coordinate box from a floating-point coordinate box
new Box ptr	**new**		Creates an empty (invalid) box
new Box ptr	**new**	(int w)	Creates a square with the given dimensions centered around the origin
new Box ptr	**new**	(int w, int h)	Creates a rectangle with given width and height, centered around the origin
new Box ptr	**new**	(int left, int bottom, int right, int top)	Creates a box with four coordinates
new Box ptr	**new**	(const Point lower_left, const Point upper_right)	Creates a box from two points

Public methods

[const]	bool	**!=**	(const Box box)	Returns true if this box is not equal to the other box
[const]	Box	**&**	(const Box box)	Returns the intersection of this box with another box
[const]	Box	*****	(const Box box)	Returns the convolution product from this box with another box
[const]	Box	*****	(double scale_factor)	Returns the scaled box
[const]	Box	**+**	(const Point point)	Joins box with a point
[const]	Box	**+**	(const Box box)	Joins two boxes
[const]	Box	**-**	(const Box box)	Subtraction of boxes

[const]	bool	**<**	(const Box box)	Returns true if this box is 'less' than another box
[const]	bool	**==**	(const Box box)	Returns true if this box is equal to the other box
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	double	**area**		Computes the box area
	void	**assign**	(const Box other)	Assigns another object to self
[const]	Box	**bbox**		Returns the bounding box
[const]	int	**bottom**		Gets the bottom coordinate of the box
	void	**bottom=**	(int c)	Sets the bottom coordinate of the box
[const]	Point	**center**		Gets the center of the box
[const]	bool	**contains?**	(int x, int y)	Returns true if the box contains the given point
[const]	bool	**contains?**	(const Point point)	Returns true if the box contains the given point
[const]	new Box ptr	**dup**		Creates a copy of self
[const]	bool	**empty?**		Returns a value indicating whether the box is empty
	Box	**enlarge**	(int dx, int dy)	Enlarges the box by a certain amount.
	Box	**enlarge**	(int d)	Enlarges the box by a certain amount on all sides.
	Box	**enlarge**	(const Vector enlargement)	Enlarges the box by a certain amount.
[const]	Box	**enlarged**	(int dx, int dy)	Enlarges the box by a certain amount.
[const]	Box	**enlarged**	(int d)	Enlarges the box by a certain amount on all sides.

[const]	Box	**enlarged**	(const Vector enlargement)	Returns the enlarged box.
[const]	unsigned long	**hash**		Computes a hash value
[const]	unsigned int	**height**		Gets the height of the box
[const]	bool	**inside?**	(const Box box)	Tests if this box is inside the argument box
[const]	bool	**is_point?**		Returns true, if the box is a single point
[const]	int	**left**		Gets the left coordinate of the box
	void	**left=**	(int c)	Sets the left coordinate of the box
	Box	**move**	(int dx, int dy)	Moves the box by a certain distance
	Box	**move**	(const Vector distance)	Moves the box by a certain distance
[const]	Box	**moved**	(int dx, int dy)	Moves the box by a certain distance
[const]	Box	**moved**	(const Vector distance)	Returns the box moved by a certain distance
[const]	bool	**overlaps?**	(const Box box)	Tests if this box overlaps the argument box
[const]	Point	**p1**		Gets the lower left point of the box
	void	**p1=**	(const Point p)	Sets the lower left point of the box
[const]	Point	**p2**		Gets the upper right point of the box
	void	**p2=**	(const Point p)	Sets the upper right point of the box
[const]	unsigned long	**perimeter**		Returns the perimeter of the box
[const]	int	**right**		Gets the right coordinate of the box
	void	**right=**	(int c)	Sets the right coordinate of the box
[const]	DBox	**to_dtype**	(double dbu = 1)	Converts the box to a floating-point coordinate box
[const]	string	**to_s**	(double dbu = 0)	Returns a string representing this box
[const]	int	**top**		Gets the top coordinate of the box
	void	**top=**	(int c)	Sets the top coordinate of the box
[const]	bool	**touches?**	(const Box box)	Tests if this box touches the argument box
[const]	Box	**transformed**	(const ICplxTrans t)	Transforms the box with the given complex transformation

[const]	Box	**transformed**	(const Trans t)	Returns the box transformed with the given simple transformation
[const]	DBox	**transformed**	(const CplxTrans t)	Returns the box transformed with the given complex transformation
[const]	unsigned int	**width**		Gets the width of the box

Public static methods and constants

new Box ptr	**from_s**	(string s)	Creates a box object from a string
Box	**world**		Gets the 'world' box

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new Box ptr	**from_dbox**	(const DBox dbox)	Use of this method is deprecated. Use new instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const Box box)

Description: Returns true if this box is not equal to the other box

Returns true, if this box and the given box are not equal

&

Signature: *[const]* Box **&** (const Box box)

Description: Returns the intersection of this box with another box

box: The box to take the intersection with

Returns: The intersection box

The intersection of two boxes is the largest box common to both boxes. The intersection may be empty if both boxes to not touch. If the boxes do not overlap but touch the result may be a single line or point with an area of zero. Overwrites this box with the result.

*****

(1) Signature: *[const]* Box ***** (const Box box)

Description: Returns the convolution product from this box with another box

box: The box to convolve with this box.

Returns: The convolved box

The * operator convolves the firstbox with the one given as the second argument. The box resulting from "convolution" is the outer boundary of the union set formed by placing the second box at every point of the first. In other words, the returned box of (p1,p2)*(q1,q2) is (p1+q1,p2+q2).

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* Box ***** (double scale_factor)

Description: Returns the scaled box

scale_factor:	The scaling factor
Returns:	The scaled box

The * operator scales the box with the given factor and returns the result.

This method has been introduced in version 0.22.

Python specific notes:
This method also implements '__rmul__'.

\+

(1) Signature: *[const]* Box **+** (const Point point)

Description: Joins box with a point

point:	The point to join with this box.
Returns:	The box joined with the point

The + operator joins a point with the box. The resulting box will enclose both the original box and the point.

(2) Signature: *[const]* Box **+** (const Box box)

Description: Joins two boxes

box:	The box to join with this box.
Returns:	The joined box

The + operator joins the first box with the one given as the second argument. Joining constructs a box that encloses both boxes given. Empty boxes are neutral: they do not change another box when joining. Overwrites this box with the result.

\-

Signature: *[const]* Box **-** (const Box box)

Description: Subtraction of boxes

box:	The box to subtract from this box.
Returns:	The result box

The - operator subtracts the argument box from self. This will return the bounding box of the are covered by self, but not by argument box. Subtracting a box from itself will render an empty box. Subtracting another box from self will modify the first box only if the argument box covers one side entirely.

This feature has been introduced in version 0.29.

<

Signature: *[const]* bool **<** (const Box box)

Description: Returns true if this box is 'less' than another box

Returns true, if this box is 'less' with respect to first and second point (in this order)

==	**Signature**: *[const]* bool **==** (const Box box) **Description**: Returns true if this box is equal to the other box Returns true, if this box and the given box are equal
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
area	**Signature**: *[const]* double **area** **Description**: Computes the box area

Returns the box area or 0 if the box is empty

assign

Signature: void **assign** (const Box other)

Description: Assigns another object to self

bbox

Signature: *[const]* Box **bbox**

Description: Returns the bounding box

This method is provided for consistency of the shape API is returns the box itself.

This method has been introduced in version 0.27.

bottom

Signature: *[const]* int **bottom**

Description: Gets the bottom coordinate of the box

Python specific notes:
The object exposes a readable attribute 'bottom'. This is the getter.

bottom=

Signature: void **bottom=** (int c)

Description: Sets the bottom coordinate of the box

Python specific notes:
The object exposes a writable attribute 'bottom'. This is the setter.

center

Signature: *[const]* Point **center**

Description: Gets the center of the box

contains?

(1) Signature: *[const]* bool **contains?** (int x, int y)

Description: Returns true if the box contains the given point

Returns: true if the point is inside the box.

Tests whether a point (x, y) is inside the box. It also returns true if the point is exactly on the box contour.

(2) Signature: *[const]* bool **contains?** (const Point point)

Description: Returns true if the box contains the given point

p: The point to test against.

Returns: true if the point is inside the box.

Tests whether a point is inside the box. It also returns true if the point is exactly on the box contour.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Box ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

empty?

Signature: *[const]* bool **empty?**

Description: Returns a value indicating whether the box is empty

An empty box may be created with the default constructor for example. Such a box is neutral when combining it with other boxes and renders empty boxes if used in box intersections and false in geometrical relationship tests.

enlarge

(1) Signature: Box **enlarge** (int dx, int dy)

Description: Enlarges the box by a certain amount.

Returns: A reference to this box.

This is a convenience method which takes two values instead of a Vector object. This method has been introduced in version 0.23.

(2) Signature: Box **enlarge** (int d)

Description: Enlarges the box by a certain amount on all sides.

Returns: A reference to this box.

This is a convenience method which takes one values instead of two values. It will apply the given enlargement in both directions. This method has been introduced in version 0.28.

(3) Signature: Box **enlarge** (const Vector enlargement)

Description: Enlarges the box by a certain amount.

enlargement: The grow or shrink amount in x and y direction

Returns: A reference to this box.

Enlarges the box by x and y value specified in the vector passed. Positive values with grow the box, negative ones will shrink the box. The result may be an empty box if the box disappears. The amount specifies the grow or shrink per edge. The width and height will change by twice the amount. Does not check for coordinate overflows.

enlarged

(1) Signature: *[const]* Box **enlarged** (int dx, int dy)

Description: Enlarges the box by a certain amount.

Returns: The enlarged box.

This is a convenience method which takes two values instead of a Vector object. This method has been introduced in version 0.23.

(2) Signature: *[const]* Box **enlarged** (int d)

Description: Enlarges the box by a certain amount on all sides.

Returns: The enlarged box.

This is a convenience method which takes one values instead of two values. It will apply the given enlargement in both directions. This method has been introduced in version 0.28.

(3) Signature: *[const]* Box **enlarged** (const Vector enlargement)

Description: Returns the enlarged box.

enlargement: The grow or shrink amount in x and y direction

Returns: The enlarged box.

Enlarges the box by x and y value specified in the vector passed. Positive values with grow the box, negative ones will shrink the box. The result may be an empty box if the box disappears. The amount specifies the grow or shrink per edge. The width and height will change by twice the amount. Does not modify this box. Does not check for coordinate overflows.

from_dbox

Signature: *[static]* new Box ptr **from_dbox** (const DBox dbox)

Description: Creates an integer coordinate box from a floating-point coordinate box

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dbox'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new Box ptr **from_s** (string s)

Description: Creates a box object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given box. This method enables boxes as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

height

Signature: *[const]* unsigned int **height**

Description: Gets the height of the box

inside?

Signature: *[const]* bool **inside?** (const Box box)

Description: Tests if this box is inside the argument box

Returns true, if this box is inside the given box, i.e. the box intersection renders this box

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_point?

Signature: *[const]* bool **is_point?**

Description: Returns true, if the box is a single point

left

Signature: *[const]* int **left**

Description: Gets the left coordinate of the box

Python specific notes:
The object exposes a readable attribute 'left'. This is the getter.

left=

Signature: void **left=** (int c)

Description: Sets the left coordinate of the box

Python specific notes:
The object exposes a writable attribute 'left'. This is the setter.

move

(1) Signature: Box **move** (int dx, int dy)

Description: Moves the box by a certain distance

Returns: A reference to this box.

This is a convenience method which takes two values instead of a Point object. This method has been introduced in version 0.23.

(2) Signature: Box **move** (const Vector distance)

Description: Moves the box by a certain distance

distance: The offset to move the box.

Returns: A reference to this box.

Moves the box by a given offset and returns the moved box. Does not check for coordinate overflows.

moved

(1) Signature: *[const]* Box **moved** (int dx, int dy)

Description: Moves the box by a certain distance

Returns: The moved box.

This is a convenience method which takes two values instead of a Point object. This method has been introduced in version 0.23.

(2) Signature: *[const]* Box **moved** (const Vector distance)

Description: Returns the box moved by a certain distance

distance: The offset to move the box.

Returns: The moved box.

Moves the box by a given offset and returns the moved box. Does not modify this box. Does not check for coordinate overflows.

new

(1) Signature: *[static]* new Box ptr **new** (const DBox dbox)

Description: Creates an integer coordinate box from a floating-point coordinate box

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dbox'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Box ptr **new**

Description: Creates an empty (invalid) box

Empty boxes don't modify a box when joined with it. The intersection between an empty and any other box is also an empty box. The width, height, p1 and p2 attributes of an empty box are undefined. Use empty? to get a value indicating whether the box is empty.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Box ptr **new** (int w)

Description: Creates a square with the given dimensions centered around the origin

Note that for integer-unit boxes, the dimension has to be an even number to avoid rounding.

This convenience constructor has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Box ptr **new** (int w, int h)

Description: Creates a rectangle with given width and height, centered around the origin

Note that for integer-unit boxes, the dimensions have to be an even number to avoid rounding.

This convenience constructor has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new Box ptr **new** (int left, int bottom, int right, int top)

Description: Creates a box with four coordinates

Four coordinates are given to create a new box. If the coordinates are not provided in the correct order (i.e. right < left), these are swapped.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new Box ptr **new** (const Point lower_left, const Point upper_right)

Description: Creates a box from two points

Two points are given to create a new box. If the coordinates are not provided in the correct order (i.e. right < left), these are swapped.

Python specific notes:
This method is the default initializer of the object.

overlaps?

Signature: *[const]* bool **overlaps?** (const Box box)

Description: Tests if this box overlaps the argument box

Returns true, if the intersection box of this box with the argument box exists and has a non-vanishing area

p1

Signature: *[const]* Point **p1**

Description: Gets the lower left point of the box

Python specific notes:
The object exposes a readable attribute 'p1'. This is the getter.

p1=

Signature: void **p1=** (const Point p)

Description: Sets the lower left point of the box

Python specific notes:
The object exposes a writable attribute 'p1'. This is the setter.

p2

Signature: *[const]* Point **p2**

Description: Gets the upper right point of the box

Python specific notes:
The object exposes a readable attribute 'p2'. This is the getter.

p2=

Signature: void **p2=** (const Point p)

Description: Sets the upper right point of the box

Python specific notes:
The object exposes a writable attribute 'p2'. This is the setter.

perimeter

Signature: *[const]* unsigned long **perimeter**

Description: Returns the perimeter of the box

This method is equivalent to 2*(width+height). For empty boxes, this method returns 0.

This method has been introduced in version 0.23.

right

Signature: *[const]* int **right**

Description: Gets the right coordinate of the box

Python specific notes:
The object exposes a readable attribute 'right'. This is the getter.

right=

Signature: void **right=** (int c)

Description: Sets the right coordinate of the box

Python specific notes:
The object exposes a writable attribute 'right'. This is the setter.

to_dtype

Signature: *[const]* DBox **to_dtype** (double dbu = 1)

Description: Converts the box to a floating-point coordinate box

The database unit can be specified to translate the integer-coordinate box into a floating-point coordinate box in micron units. The database unit is basically a scaling factor.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: Returns a string representing this box

This string can be turned into a box again by using from_s . If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

top

Signature: *[const]* int **top**

Description: Gets the top coordinate of the box

Python specific notes:
The object exposes a readable attribute 'top'. This is the getter.

top=

Signature: void **top=** (int c)

Description: Sets the top coordinate of the box

Python specific notes:
The object exposes a writable attribute 'top'. This is the setter.

touches?

Signature: *[const]* bool **touches?** (const Box box)

Description: Tests if this box touches the argument box

Two boxes touch if they overlap or their boundaries share at least one common point. Touching is equivalent to a non-empty intersection ('!(b1 & b2).empty?').

transformed

(1) Signature: *[const]* Box **transformed** (const ICplxTrans t)

Description: Transforms the box with the given complex transformation

t:	The magnifying transformation to apply
Returns:	The transformed box (in this case an integer coordinate box)

This method has been introduced in version 0.18.

(2) Signature: *[const]* Box **transformed** (const Trans t)

Description: Returns the box transformed with the given simple transformation

t:	The transformation to apply
Returns:	The transformed box

(3) Signature: *[const]* DBox **transformed** (const CplxTrans t)

Description: Returns the box transformed with the given complex transformation

t:	The magnifying transformation to apply
Returns:	The transformed box (a DBox now)

width

Signature: *[const]* unsigned int **width**

Description: Gets the width of the box

world

Signature: *[static]* Box **world**

Description: Gets the 'world' box

The world box is the biggest box that can be represented. So it is basically 'all'. The world box behaves neutral on intersections for example. In other operations such as displacement or transformations, the world box may render unexpected results because of coordinate overflow.

The world box can be used

- for comparison ('==', '!=', '<')
- in union and intersection ('+' and '&')
- in relations (contains?, overlaps?, touches?)
- as 'all' argument in region queries

This method has been introduced in version 0.28.

4.18. API reference - Class DBox

Notation used in Ruby API documentation

Module: db

Description: A box class with floating-point coordinates

This object represents a box (a rectangular shape).

The definition of the attributes is: p1 is the lower left point, p2 the upper right one. If a box is constructed from two points (or four coordinates), the coordinates are sorted accordingly.

A box can be empty. An empty box represents no area (not even a point). Empty boxes behave neutral with respect to most operations. Empty boxes return true on empty?.

A box can be a point or a single line. In this case, the area is zero but the box still can overlap other boxes for example and it is not empty.

See The Database API for more details about the database objects.

Public constructors

new DBox ptr	**new**	(const Box box)	Creates a floating-point coordinate box from an integer coordinate box
new DBox ptr	**new**		Creates an empty (invalid) box
new DBox ptr	**new**	(double w)	Creates a square with the given dimensions centered around the origin
new DBox ptr	**new**	(double w, double h)	Creates a rectangle with given width and height, centered around the origin
new DBox ptr	**new**	(double left, double bottom, double right, double top)	Creates a box with four coordinates
new DBox ptr	**new**	(const DPoint lower_left, const DPoint upper_right)	Creates a box from two points

Public methods

[const]	bool	**!=**	(const DBox box)	Returns true if this box is not equal to the other box
[const]	DBox	**&**	(const DBox box)	Returns the intersection of this box with another box
[const]	DBox	*****	(const DBox box)	Returns the convolution product from this box with another box
[const]	DBox	*****	(double scale_factor)	Returns the scaled box
[const]	DBox	**+**	(const DPoint point)	Joins box with a point
[const]	DBox	**+**	(const DBox box)	Joins two boxes
[const]	DBox	**-**	(const DBox box)	Subtraction of boxes

[const]	bool	**<**	(const DBox box)	Returns true if this box is 'less' than another box
[const]	bool	**==**	(const DBox box)	Returns true if this box is equal to the other box
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	double	**area**		Computes the box area
	void	**assign**	(const DBox other)	Assigns another object to self
[const]	DBox	**bbox**		Returns the bounding box
[const]	double	**bottom**		Gets the bottom coordinate of the box
	void	**bottom=**	(double c)	Sets the bottom coordinate of the box
[const]	DPoint	**center**		Gets the center of the box
[const]	bool	**contains?**	(double x, double y)	Returns true if the box contains the given point
[const]	bool	**contains?**	(const DPoint point)	Returns true if the box contains the given point
[const]	new DBox ptr	**dup**		Creates a copy of self
[const]	bool	**empty?**		Returns a value indicating whether the box is empty
	DBox	**enlarge**	(double dx, double dy)	Enlarges the box by a certain amount.
	DBox	**enlarge**	(double d)	Enlarges the box by a certain amount on all sides.
	DBox	**enlarge**	(const DVector enlargement)	Enlarges the box by a certain amount.
[const]	DBox	**enlarged**	(double dx, double dy)	Enlarges the box by a certain amount.

[const]	DBox	**enlarged**	(double d)	Enlarges the box by a certain amount on all sides.
[const]	DBox	**enlarged**	(const DVector enlargement)	Returns the enlarged box.
[const]	unsigned long	**hash**		Computes a hash value
[const]	double	**height**		Gets the height of the box
[const]	bool	**inside?**	(const DBox box)	Tests if this box is inside the argument box
[const]	bool	**is_point?**		Returns true, if the box is a single point
[const]	double	**left**		Gets the left coordinate of the box
	void	**left=**	(double c)	Sets the left coordinate of the box
	DBox	**move**	(double dx, double dy)	Moves the box by a certain distance
	DBox	**move**	(const DVector distance)	Moves the box by a certain distance
[const]	DBox	**moved**	(double dx, double dy)	Moves the box by a certain distance
[const]	DBox	**moved**	(const DVector distance)	Returns the box moved by a certain distance
[const]	bool	**overlaps?**	(const DBox box)	Tests if this box overlaps the argument box
[const]	DPoint	**p1**		Gets the lower left point of the box
	void	**p1=**	(const DPoint p)	Sets the lower left point of the box
[const]	DPoint	**p2**		Gets the upper right point of the box
	void	**p2=**	(const DPoint p)	Sets the upper right point of the box
[const]	double	**perimeter**		Returns the perimeter of the box
[const]	double	**right**		Gets the right coordinate of the box
	void	**right=**	(double c)	Sets the right coordinate of the box
[const]	Box	**to_itype**	(double dbu = 1)	Converts the box to an integer coordinate box
[const]	string	**to_s**	(double dbu = 0)	Returns a string representing this box
[const]	double	**top**		Gets the top coordinate of the box
	void	**top=**	(double c)	Sets the top coordinate of the box
[const]	bool	**touches?**	(const DBox box)	Tests if this box touches the argument box

[const]	Box	**transformed**	(const VCplxTrans t)	Transforms the box with the given complex transformation
[const]	DBox	**transformed**	(const DTrans t)	Returns the box transformed with the given simple transformation
[const]	DBox	**transformed**	(const DCplxTrans t)	Returns the box transformed with the given complex transformation
[const]	double	**width**		Gets the width of the box

Public static methods and constants

	new DBox ptr	**from_s**	(string s)	Creates a box object from a string
	DBox	**world**		Gets the 'world' box

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new DBox ptr	**from_ibox**	(const Box box)	Use of this method is deprecated. Use new instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const DBox box)

Description: Returns true if this box is not equal to the other box

Returns true, if this box and the given box are not equal

&

Signature: *[const]* DBox **&** (const DBox box)

Description: Returns the intersection of this box with another box

box: The box to take the intersection with

Returns: The intersection box

The intersection of two boxes is the largest box common to both boxes. The intersection may be empty if both boxes to not touch. If the boxes do not overlap but touch the result may be a single line or point with an area of zero. Overwrites this box with the result.

*****

(1) Signature: *[const]* DBox ***** (const DBox box)

Description: Returns the convolution product from this box with another box

box: The box to convolve with this box.

Returns: The convolved box

The * operator convolves the firstbox with the one given as the second argument. The box resulting from "convolution" is the outer boundary of the union set formed by placing the second box at every point of the first. In other words, the returned box of (p1,p2)*(q1,q2) is (p1+q1,p2+q2).

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* DBox ***** (double scale_factor)

Description: Returns the scaled box

scale_factor: The scaling factor

Returns: The scaled box

The * operator scales the box with the given factor and returns the result.

This method has been introduced in version 0.22.

Python specific notes:
This method also implements '__rmul__'.

\+

(1) Signature: *[const]* DBox **+** (const DPoint point)

Description: Joins box with a point

point: The point to join with this box.

Returns: The box joined with the point

The + operator joins a point with the box. The resulting box will enclose both the original box and the point.

(2) Signature: *[const]* DBox **+** (const DBox box)

Description: Joins two boxes

box: The box to join with this box.

Returns: The joined box

The + operator joins the first box with the one given as the second argument. Joining constructs a box that encloses both boxes given. Empty boxes are neutral: they do not change another box when joining. Overwrites this box with the result.

\-

Signature: *[const]* DBox **-** (const DBox box)

Description: Subtraction of boxes

box: The box to subtract from this box.

Returns: The result box

The - operator subtracts the argument box from self. This will return the bounding box of the are covered by self, but not by argument box. Subtracting a box from itself will render an empty box. Subtracting another box from self will modify the first box only if the argument box covers one side entirely.

This feature has been introduced in version 0.29.

<

Signature: *[const]* bool **<** (const DBox box)

Description: Returns true if this box is 'less' than another box

	Returns true, if this box is 'less' with respect to first and second point (in this order)
==	**Signature**: *[const]* bool **==** (const DBox box) **Description**: Returns true if this box is equal to the other box Returns true, if this box and the given box are equal
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
area	**Signature**: *[const]* double **area** **Description**: Computes the box area

Returns the box area or 0 if the box is empty

assign

Signature: void **assign** (const DBox other)

Description: Assigns another object to self

bbox

Signature: *[const]* DBox **bbox**

Description: Returns the bounding box

This method is provided for consistency of the shape API is returns the box itself.

This method has been introduced in version 0.27.

bottom

Signature: *[const]* double **bottom**

Description: Gets the bottom coordinate of the box

Python specific notes:
The object exposes a readable attribute 'bottom'. This is the getter.

bottom=

Signature: void **bottom=** (double c)

Description: Sets the bottom coordinate of the box

Python specific notes:
The object exposes a writable attribute 'bottom'. This is the setter.

center

Signature: *[const]* DPoint **center**

Description: Gets the center of the box

contains?

(1) Signature: *[const]* bool **contains?** (double x, double y)

Description: Returns true if the box contains the given point

Returns: true if the point is inside the box.

Tests whether a point (x, y) is inside the box. It also returns true if the point is exactly on the box contour.

(2) Signature: *[const]* bool **contains?** (const DPoint point)

Description: Returns true if the box contains the given point

p: The point to test against.

Returns: true if the point is inside the box.

Tests whether a point is inside the box. It also returns true if the point is exactly on the box contour.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new DBox ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

empty?

Signature: *[const]* bool **empty?**

Description: Returns a value indicating whether the box is empty

An empty box may be created with the default constructor for example. Such a box is neutral when combining it with other boxes and renders empty boxes if used in box intersections and false in geometrical relationship tests.

enlarge

(1) Signature: DBox **enlarge** (double dx, double dy)

Description: Enlarges the box by a certain amount.

Returns: A reference to this box.

This is a convenience method which takes two values instead of a Vector object. This method has been introduced in version 0.23.

(2) Signature: DBox **enlarge** (double d)

Description: Enlarges the box by a certain amount on all sides.

Returns: A reference to this box.

This is a convenience method which takes one values instead of two values. It will apply the given enlargement in both directions. This method has been introduced in version 0.28.

(3) Signature: DBox **enlarge** (const DVector enlargement)

Description: Enlarges the box by a certain amount.

enlargement: The grow or shrink amount in x and y direction

Returns: A reference to this box.

Enlarges the box by x and y value specified in the vector passed. Positive values with grow the box, negative ones will shrink the box. The result may be an empty box if the box disappears. The amount specifies the grow or shrink per edge. The width and height will change by twice the amount. Does not check for coordinate overflows.

enlarged

(1) Signature: *[const]* DBox **enlarged** (double dx, double dy)

Description: Enlarges the box by a certain amount.

Returns: The enlarged box.

This is a convenience method which takes two values instead of a Vector object. This method has been introduced in version 0.23.

(2) Signature: *[const]* DBox **enlarged** (double d)

Description: Enlarges the box by a certain amount on all sides.

Returns: The enlarged box.

This is a convenience method which takes one values instead of two values. It will apply the given enlargement in both directions. This method has been introduced in version 0.28.

(3) Signature: *[const]* DBox **enlarged** (const DVector enlargement)

Description: Returns the enlarged box.

enlargement: The grow or shrink amount in x and y direction

Returns: The enlarged box.

Enlarges the box by x and y value specified in the vector passed. Positive values with grow the box, negative ones will shrink the box. The result may be an empty box if the box disappears. The amount specifies the grow or shrink per edge. The width and height will change by twice the amount. Does not modify this box. Does not check for coordinate overflows.

from_ibox

Signature: *[static]* new DBox ptr **from_ibox** (const Box box)

Description: Creates a floating-point coordinate box from an integer coordinate box

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_ibox'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new DBox ptr **from_s** (string s)

Description: Creates a box object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given box. This method enables boxes as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

height

Signature: *[const]* double **height**

Description: Gets the height of the box

inside?

Signature: *[const]* bool **inside?** (const DBox box)

Description: Tests if this box is inside the argument box

Returns true, if this box is inside the given box, i.e. the box intersection renders this box

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_point?

Signature: *[const]* bool **is_point?**

Description: Returns true, if the box is a single point

left

Signature: *[const]* double **left**

Description: Gets the left coordinate of the box

Python specific notes:
The object exposes a readable attribute 'left'. This is the getter.

left=

Signature: void **left=** (double c)

Description: Sets the left coordinate of the box

Python specific notes:
The object exposes a writable attribute 'left'. This is the setter.

move

(1) Signature: DBox **move** (double dx, double dy)

Description: Moves the box by a certain distance

Returns: A reference to this box.

This is a convenience method which takes two values instead of a Point object. This method has been introduced in version 0.23.

(2) Signature: DBox **move** (const DVector distance)

Description: Moves the box by a certain distance

distance: The offset to move the box.

Returns: A reference to this box.

Moves the box by a given offset and returns the moved box. Does not check for coordinate overflows.

moved

(1) Signature: *[const]* DBox **moved** (double dx, double dy)

Description: Moves the box by a certain distance

Returns: The moved box.

This is a convenience method which takes two values instead of a Point object. This method has been introduced in version 0.23.

(2) Signature: *[const]* DBox **moved** (const DVector distance)

Description: Returns the box moved by a certain distance

distance: The offset to move the box.

Returns: The moved box.

Moves the box by a given offset and returns the moved box. Does not modify this box. Does not check for coordinate overflows.

new

(1) Signature: *[static]* new DBox ptr **new** (const Box box)

Description: Creates a floating-point coordinate box from an integer coordinate box

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_ibox'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DBox ptr **new**

Description: Creates an empty (invalid) box

Empty boxes don't modify a box when joined with it. The intersection between an empty and any other box is also an empty box. The width, height, p1 and p2 attributes of an empty box are undefined. Use empty? to get a value indicating whether the box is empty.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new DBox ptr **new** (double w)

Description: Creates a square with the given dimensions centered around the origin

Note that for integer-unit boxes, the dimension has to be an even number to avoid rounding.

This convenience constructor has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new DBox ptr **new** (double w, double h)

Description: Creates a rectangle with given width and height, centered around the origin

Note that for integer-unit boxes, the dimensions have to be an even number to avoid rounding.

This convenience constructor has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new DBox ptr **new** (double left, double bottom, double right, double top)

Description: Creates a box with four coordinates

Four coordinates are given to create a new box. If the coordinates are not provided in the correct order (i.e. right < left), these are swapped.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new DBox ptr **new** (const DPoint lower_left, const DPoint upper_right)

Description: Creates a box from two points

Two points are given to create a new box. If the coordinates are not provided in the correct order (i.e. right < left), these are swapped.

Python specific notes:
This method is the default initializer of the object.

overlaps?

Signature: *[const]* bool **overlaps?** (const DBox box)

Description: Tests if this box overlaps the argument box

Returns true, if the intersection box of this box with the argument box exists and has a non-vanishing area

p1

Signature: *[const]* DPoint **p1**

Description: Gets the lower left point of the box

Python specific notes:
The object exposes a readable attribute 'p1'. This is the getter.

p1=

Signature: void **p1=** (const DPoint p)

Description: Sets the lower left point of the box

Python specific notes:
The object exposes a writable attribute 'p1'. This is the setter.

p2

Signature: *[const]* DPoint **p2**

Description: Gets the upper right point of the box

Python specific notes:
The object exposes a readable attribute 'p2'. This is the getter.

p2=

Signature: void **p2=** (const DPoint p)

Description: Sets the upper right point of the box

Python specific notes:
The object exposes a writable attribute 'p2'. This is the setter.

perimeter

Signature: *[const]* double **perimeter**

Description: Returns the perimeter of the box

This method is equivalent to 2*(width+height). For empty boxes, this method returns 0.

This method has been introduced in version 0.23.

right

Signature: *[const]* double **right**

Description: Gets the right coordinate of the box

Python specific notes:
The object exposes a readable attribute 'right'. This is the getter.

right=

Signature: void **right=** (double c)

Description: Sets the right coordinate of the box

Python specific notes:
The object exposes a writable attribute 'right'. This is the setter.

to_itype

Signature: *[const]* Box **to_itype** (double dbu = 1)

Description: Converts the box to an integer coordinate box

The database unit can be specified to translate the floating-point coordinate box in micron units to an integer-coordinate box in database units. The boxes coordinates will be divided by the database unit.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: Returns a string representing this box

This string can be turned into a box again by using from_s . If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

top

Signature: *[const]* double **top**

Description: Gets the top coordinate of the box

Python specific notes:
The object exposes a readable attribute 'top'. This is the getter.

top=

Signature: void **top=** (double c)

Description: Sets the top coordinate of the box

Python specific notes:
The object exposes a writable attribute 'top'. This is the setter.

touches?

Signature: *[const]* bool **touches?** (const DBox box)

Description: Tests if this box touches the argument box

Two boxes touch if they overlap or their boundaries share at least one common point. Touching is equivalent to a non-empty intersection ('!(b1 & b2).empty?').

transformed

(1) Signature: *[const]* Box **transformed** (const VCplxTrans t)

Description: Transforms the box with the given complex transformation

t: The magnifying transformation to apply

Returns: The transformed box (in this case an integer coordinate box)

This method has been introduced in version 0.25.

(2) Signature: *[const]* DBox **transformed** (const DTrans t)

Description: Returns the box transformed with the given simple transformation

t: The transformation to apply

Returns: The transformed box

(3) Signature: *[const]* DBox **transformed** (const DCplxTrans t)

Description: Returns the box transformed with the given complex transformation

t: The magnifying transformation to apply

Returns: The transformed box (a DBox now)

width

Signature: *[const]* double **width**

Description: Gets the width of the box

world

Signature: *[static]* DBox **world**

Description: Gets the 'world' box

The world box is the biggest box that can be represented. So it is basically 'all'. The world box behaves neutral on intersections for example. In other operations such as displacement or transformations, the world box may render unexpected results because of coordinate overflow.

The world box can be used

- for comparison ('==', '!=', '<')
- in union and intersection ('+' and '&')
- in relations (contains?, overlaps?, touches?)
- as 'all' argument in region queries

This method has been introduced in version 0.28.

4.19. API reference - Class Cell

Notation used in Ruby API documentation

Module: db

Description: A cell

A cell object consists of a set of shape containers (called layers), a set of child cell instances and auxiliary information such as the parent instance list. A cell is identified through an index given to the cell upon instantiation. Cell instances refer to single instances or array instances. Both are encapsulated in the same object, the CellInstArray object. In the simple case, this object refers to a single instance. In the general case, this object may refer to a regular array of cell instances as well.

Starting from version 0.16, the child_inst and erase_inst methods are no longer available since they were using index addressing which is no longer supported. Instead, instances are now addressed with the Instance reference objects.

See The Database API for more details about the database objects like the Cell class.

Public constructors

new Cell ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add_meta_info**	(const LayoutMetaInfo info)	Adds meta information to the cell
[const]	string	**basic_name**		Returns the name of the library or PCell or the real name of the cell
[const]	Box	**bbox**		Gets the bounding box of the cell
[const]	Box	**bbox**	(unsigned int layer_index)	Gets the per-layer bounding box of the cell
[const]	RecursiveInstanceIt	**begin_instances_rec**		Delivers a recursive instance iterator for the instances below the cell
[const]	RecursiveInstanceIterator	**begin_instances_rec_overlapping**	(DBox region)	Delivers a recursive instance iterator for the instances below the cell using a region search
[const]	RecursiveInstanceIt	**begin_instances_rec**	(DBox region)	Delivers a recursive instance iterator for the instances below the cell using a region

				search, with the region given in micrometer units
[const]	RecursiveInstanceIterator	**begin_instances_rec_touching**	(DBox region)	Delivers a recursive instance iterator for the instances below the cell
[const]	RecursiveInstanceIt	**begin_instances_rec_**	(DBox region)	Delivers a recursive instance iterator for the instances below the cell using a region search, with the region given in micrometer units
[const]	RecursiveShapeIterator	**begin_shapes_rec**	(unsigned int layer)	Delivers a recursive shape iterator for the shapes below the cell on the given layer
[const]	RecursiveShapeIter	**begin_shapes_rec_ov**	(unsigned int layer, Box region)	Delivers a recursive shape iterator for the shapes below the cell on the given layer using a region search
[const]	RecursiveShapeIterator	**begin_shapes_rec_overlapping**	(unsigned int layer, DBox region)	Delivers a recursive shape iterator for the shapes below the cell on the given layer using a region search, with the region given in micrometer units
[const]	RecursiveShapeIter	**begin_shapes_rec_to**	(unsigned int layer, Box region)	Delivers a recursive shape iterator for the shapes below the cell on the given layer using a region search
[const]	RecursiveShapeIterator	**begin_shapes_rec_touching**	(unsigned int layer, DBox region)	Delivers a recursive shape iterator for the shapes below the cell on the given layer using a region search, with the region given in micrometer units
[const]	unsigned int[]	**called_cells**		Gets a list of all called cells
[const]	unsigned int[]	**caller_cells**		Gets a list of all caller cells
[const]	unsigned int	**cell_index**		Gets the cell index
	Instance	**change_pcell_parameter**	(const Instance instance, string name, variant value)	Changes a single parameter for an individual PCell instance given by name
	Instance	**change_pcell_parame**	(const Instance instance, map<string,variant> dict)	Changes the given parameter for an individual PCell instance
	Instance	**change_pcell_parameters**	(const Instance instance, variant[] parameters)	Changes the parameters for an individual PCell instance
[const]	unsigned long	**child_cells**		Gets the number of child cells
[const]	unsigned long	**child_instances**		Gets the number of child instances
	void	**clear**	(unsigned int layer_index)	Clears the shapes on the given layer

	void	clear		Clears the cell (deletes shapes and instances)
	void	clear_insts		Clears the instance list
	void	clear_meta_info		Clears the meta information of the cell
	void	clear_shapes		Clears all shapes in the cell
	void	copy	(unsigned int src, unsigned int dest)	Copies the shapes from the source to the target layer
	void	copy	(Cell ptr src_cell, unsigned int src_layer, unsigned int dest)	Copies shapes from another cell to the target layer in this cell
	void	copy_instances	(const Cell source_cell)	Copies the instances of child cells in the source cell to this cell
	void	copy_meta_info	(const Cell ptr other)	Copies the meta information from the other cell into this cell
	void	copy_shapes	(const Cell source_cell)	Copies the shapes from the given cell into this cell
	void	copy_shapes	(const Cell source_cell, const LayerMapping layer_mapping)	Copies the shapes from the given cell into this cell
	unsigned int[]	copy_tree	(const Cell source_cell)	Copies the cell tree of the given cell into this cell
	void	copy_tree_shapes	(const Cell source_cell, const CellMapping cell_mapping)	Copies the shapes from the given cell and the cell tree below into this cell or subcells of this cell
	void	copy_tree_shapes	(const Cell source_cell, const CellMapping cell_mapping, const LayerMapping layer_mapping)	Copies the shapes from the given cell and the cell tree below into this cell or subcells of this cell with layer mapping
[const]	DBox	dbbox		Gets the bounding box of the cell in micrometer units
[const]	DBox	dbbox	(unsigned int layer_index)	Gets the per-layer bounding box of the cell in micrometer units
	void	delete		Deletes this cell
	void	delete_property	(variant key)	Deletes the user property with the given key
[const]	string	display_title		Returns a nice looking name for display purposes
[const]	Cell ptr	dup		Creates a copy of the cell
[const,iter]	unsigned int	each_child_cell		Iterates over all child cells

[iter]	Instance	each_inst		Iterates over all child instances (which may actually be instance arrays)
[const,iter]	LayoutMetaInfo	each_meta_info		Iterates over the meta information of the cell
[const,iter]	Instance	each_overlapping_inst	(const Box b)	Gets the instances overlapping the given rectangle
[const,iter]	Instance	each_overlapping_ins	(const DBox b)	Gets the instances overlapping the given rectangle, with the rectangle in micrometer units
[const,iter]	Shape	each_overlapping_shape	(unsigned int layer_index, const Box box, unsigned int flags)	Iterates over all shapes of a given layer that overlap the given box
[const,iter]	Shape	each_overlapping_sh	(unsigned int layer_index, const Box box)	Iterates over all shapes of a given layer that overlap the given box
[const,iter]	Shape	each_overlapping_shape	(unsigned int layer_index, const DBox box, unsigned int flags)	Iterates over all shapes of a given layer that overlap the given box, with the box given in micrometer units
[const,iter]	Shape	each_overlapping_sh	(unsigned int layer_index, const DBox box)	Iterates over all shapes of a given layer that overlap the given box, with the box given in micrometer units
[const,iter]	unsigned int	each_parent_cell		Iterates over all parent cells
[const,iter]	ParentInstArray	each_parent_inst		Iterates over the parent instance list (which may actually be instance arrays)
[const,iter]	Shape	each_shape	(unsigned int layer_index, unsigned int flags)	Iterates over all shapes of a given layer
[const,iter]	Shape	each_shape	(unsigned int layer_index)	Iterates over all shapes of a given layer
[const,iter]	Instance	each_touching_inst	(const Box b)	Gets the instances touching the given rectangle
[const,iter]	Instance	each_touching_inst	(const DBox b)	Gets the instances touching the given rectangle, with the rectangle in micrometer units
[const,iter]	Shape	each_touching_shape	(unsigned int layer_index, const Box box, unsigned int flags)	Iterates over all shapes of a given layer that touch the given box
[const,iter]	Shape	each_touching_shape	(unsigned int layer_index, const Box box)	Iterates over all shapes of a given layer that touch the given box

[const,iter]	Shape	**each_touching_shape**	(unsigned int layer_index, const DBox box, unsigned int flags)	Iterates over all shapes of a given layer that touch the given box, with the box given in micrometer units
[const,iter]	Shape	**each_touching_shape**	(unsigned int layer_index, const DBox box)	Iterates over all shapes of a given layer that touch the given box, with the box given in micrometer units
	void	**erase**	(const Instance inst)	Erases the instance given by the Instance object
	void	**fill_region**	(const Region region, unsigned int fill_cell_index, const Box fc_box, const Point ptr origin = (0, 0), Region ptr remaining_parts = nil, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil, const Box glue_box = ())	Fills the given region with cells of the given type (extended version)
	void	**fill_region**	(const Region region, unsigned int fill_cell_index, const Box fc_bbox, const Vector row_step, const Vector column_step, const Point ptr origin = (0, 0), Region ptr remaining_parts = nil, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil, const Box glue_box = ())	Fills the given region with cells of the given type (skew step version)
	void	**fill_region_multi**	(const Region region, unsigned int fill_cell_index, const Box fc_bbox, const Vector row_step, const Vector column_step, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil,	Fills the given region with cells of the given type in enhanced mode with iterations

			const Box glue_box = ())	
	void	**flatten**	(bool prune)	Flattens the given cell
	void	**flatten**	(int levels, bool prune)	Flattens the given cell
	void	**ghost_cell=**	(bool flag)	Sets the "ghost cell" flag
[const]	bool	**has_prop_id?**		Returns true, if the cell has user properties
[const]	unsigned int	**hierarchy_levels**		Returns the number of hierarchy levels below
	Instance	**insert**	(const Instance inst)	Inserts a cell instance given by another reference
	Instance	**insert**	(const CellInstArray cell_inst_array)	Inserts a cell instance (array)
	Instance	**insert**	(const DCellInstArray cell_inst_array)	Inserts a cell instance (array) given in micron units
	Instance	**insert**	(const DCellInstArray cell_inst_array, unsigned long property_id)	Inserts a cell instance (array) given in micron units with properties
	Instance	**insert**	(const CellInstArray cell_inst_array, unsigned long property_id)	Inserts a cell instance (array) with properties
[const]	bool	**is_empty?**		Returns a value indicating whether the cell is empty
[const]	bool	**is_ghost_cell?**		Returns a value indicating whether the cell is a "ghost cell"
[const]	bool	**is_leaf?**		Gets a value indicating whether the cell is a leaf cell
[const]	bool	**is_library_cell?**		Returns true, if the cell is a proxy cell pointing to a library cell
[const]	bool	**is_pcell_variant?**		Returns true, if this cell is a pcell variant
[const]	bool	**is_pcell_variant?**	(const Instance instance)	Returns true, if this instance is a PCell variant
[const]	bool	**is_proxy?**		Returns true, if the cell presents some external entity
[const]	bool	**is_top?**		Gets a value indicating whether the cell is a top-level cell
[const]	bool	**is_valid?**	(const Instance instance)	Tests if the given Instance object is still pointing to a valid object

	Layout ptr	**layout**		Returns a reference to the layout where the cell resides
[const]	const Layout ptr	**layout**		Returns a reference to the layout where the cell resides (const references)
[const]	Library ptr	**library**		Returns a reference to the library from which the cell is imported
[const]	unsigned int	**library_cell_index**		Returns the index of the cell in the layout of the library (if it's a library proxy)
	void	**merge_meta_info**	(const Cell ptr other)	Merges the meta information from the other cell into this cell
	new LayoutMetaInfo ptr	**meta_info**	(string name)	Gets the meta information for a given name
	variant	**meta_info_value**	(string name)	Gets the meta information value for a given name
	void	**move**	(unsigned int src, unsigned int dest)	Moves the shapes from the source to the target layer
	void	**move**	(Cell ptr src_cell, unsigned int src_layer, unsigned int dest)	Moves shapes from another cell to the target layer in this cell
	void	**move_instances**	(Cell source_cell)	Moves the instances of child cells in the source cell to this cell
	void	**move_shapes**	(Cell source_cell)	Moves the shapes from the given cell into this cell
	void	**move_shapes**	(Cell source_cell, const LayerMapping layer_mapping)	Moves the shapes from the given cell into this cell
	unsigned int[]	**move_tree**	(Cell source_cell)	Moves the cell tree of the given cell into this cell
	void	**move_tree_shapes**	(Cell source_cell, const CellMapping cell_mapping)	Moves the shapes from the given cell and the cell tree below into this cell or subcells of this cell
	void	**move_tree_shapes**	(Cell source_cell, const CellMapping cell_mapping, const LayerMapping layer_mapping)	Moves the shapes from the given cell and the cell tree below into this cell or subcells of this cell with layer mapping
[const]	string	**name**		Gets the cell's name
	void	**name=**	(string name)	Renames the cell
[const]	unsigned long	**parent_cells**		Gets the number of parent cells

[const]	const PCellDeclaration ptr	**pcell_declaration**		Returns a reference to the PCell declaration
[const]	const PCellDeclaration ptr	**pcell_declaration**	(const Instance instance)	Returns the PCell declaration of a pcell instance
[const]	unsigned int	**pcell_id**		Returns the PCell ID if the cell is a pcell variant
[const]	Library ptr	**pcell_library**		Returns the library where the PCell is declared if this cell is a PCell and it is not defined locally.
[const]	variant	**pcell_parameter**	(string name)	Gets a PCell parameter by name if the cell is a PCell variant
[const]	variant	**pcell_parameter**	(const Instance instance, string name)	Returns a PCell parameter by name for a pcell instance
[const]	variant[]	**pcell_parameters**		Returns the PCell parameters for a pcell variant
[const]	variant[]	**pcell_parameters**	(const Instance instance)	Returns the PCell parameters for a pcell instance
[const]	map<string,variant>	**pcell_parameters_by_**		Returns the PCell parameters for a pcell variant as a name to value dictionary
[const]	map<string,variant>	**pcell_parameters_by_name**	(const Instance instance)	Returns the PCell parameters for a pcell instance as a name to value dictionary
[const]	unsigned long	**prop_id**		Gets the properties ID associated with the cell
	void	**prop_id=**	(unsigned long id)	Sets the properties ID associated with the cell
	variant	**property**	(variant key)	Gets the user property with the given key
	void	**prune_cell**		Deletes the cell plus subcells not used otherwise
	void	**prune_cell**	(int levels)	Deletes the cell plus subcells not used otherwise
	void	**prune_subcells**		Deletes all sub cells of the cell which are not used otherwise
	void	**prune_subcells**	(int levels)	Deletes all sub cells of the cell which are not used otherwise down to the specified level of hierarchy
[const]	string	**qname**		Returns the library-qualified name
	unsigned int[]	**read**	(string file_name,	Reads a layout file into this cell

			const LoadLayoutOptions options)	
	unsigned int[]	**read**	(string file_name)	Reads a layout file into this cell
	void	**refresh**		Refreshes a proxy cell
	void	**remove_meta_info**	(string name)	Removes meta information from the cell
	Instance	**replace**	(const Instance instance, const CellInstArray cell_inst_array)	Replaces a cell instance (array) with a different one
	Instance	**replace**	(const Instance instance, const CellInstArray cell_inst_array, unsigned long property_id)	Replaces a cell instance (array) with a different one with properties
	Instance	**replace**	(const Instance instance, const DCellInstArray cell_inst_array)	Replaces a cell instance (array) with a different one, given in micrometer units
	Instance	**replace**	(const Instance instance, const DCellInstArray cell_inst_array, unsigned long property_id)	Replaces a cell instance (array) with a different one and new properties, where the cell instance is given in micrometer units
	Instance	**replace_prop_id**	(const Instance instance, unsigned long property_id)	Replaces (or install) the properties of a cell
	void	**set_property**	(variant key, variant value)	Sets the user property with the given key to the given value
	Shapes	**shapes**	(unsigned int layer_index)	Returns the shapes list of the given layer
[const]	const Shapes ptr	**shapes**	(unsigned int layer_index)	Returns the shapes list of the given layer (const version)
	void	**swap**	(unsigned int layer_index1, unsigned int layer_index2)	Swaps the layers given
	Instance	**transform**	(const Instance instance, const Trans trans)	Transforms the instance with the given transformation
	Instance	**transform**	(const Instance instance,	Transforms the instance with the given complex integer transformation

			const ICplxTrans trans)	
	Instance	**transform**	(const Instance instance, const DTrans trans)	Transforms the instance with the transformation given in micrometer units
	Instance	**transform**	(const Instance instance, const DCplxTrans trans)	Transforms the instance with the given complex floating-point transformation given in micrometer units
	void	**transform**	(const Trans trans)	Transforms the cell by the given integer transformation
	void	**transform**	(const ICplxTrans trans)	Transforms the cell by the given complex integer transformation
	void	**transform**	(const DTrans trans)	Transforms the cell by the given, micrometer-unit transformation
	void	**transform**	(const DCplxTrans trans)	Transforms the cell by the given, micrometer-unit transformation
	Instance	**transform_into**	(const Instance instance, const Trans trans)	Transforms the instance into a new coordinate system with the given transformation
	Instance	**transform_into**	(const Instance instance, const ICplxTrans trans)	Transforms the instance into a new coordinate system with the given complex integer transformation
	void	**transform_into**	(const Trans trans)	Transforms the cell into a new coordinate system with the given transformation
	void	**transform_into**	(const ICplxTrans trans)	Transforms the cell into a new coordinate system with the given complex integer transformation
	Instance	**transform_into**	(const Instance instance, const DTrans trans)	Transforms the instance into a new coordinate system with the given transformation where the transformation is in micrometer units
	Instance	**transform_into**	(const Instance instance, const DCplxTrans trans)	Transforms the instance into a new coordinate system with the given complex transformation where the transformation is in micrometer units
	void	**transform_into**	(const DTrans trans)	Transforms the cell into a new coordinate system with the given transformation where the transformation is in micrometer units
	void	**transform_into**	(const DCplxTrans trans)	Transforms the cell into a new coordinate system with the given complex integer transformation where the transformation is in micrometer units
[const]	void	**write**	(string file_name)	Writes the cell to a layout file
[const]	void	**write**	(string file_name,	Writes the cell to a layout file

			const SaveLayoutOptions options)	

Deprecated methods (protected, public, static, non-static and constructors)

[const]	Box	**bbox_per_layer**	(unsigned int layer_index)	Use of this method is deprecated. Use bbox instead
	void	**create**		Use of this method is deprecated. Use _create instead
[const]	DBox	**dbbox_per_layer**	(unsigned int layer_index)	Use of this method is deprecated. Use dbbox instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_meta_info

Signature: void **add_meta_info** (const LayoutMetaInfo info)

Description: Adds meta information to the cell

See LayoutMetaInfo for details about cells and meta information.

This method has been introduced in version 0.28.8.

basic_name

Signature: *[const]* string **basic_name**

Description: Returns the name of the library or PCell or the real name of the cell

For non-proxy cells (see is_proxy?), this method simply returns the cell name. For proxy cells, this method returns the PCells definition name or the library cell name. This name may differ from the actual cell's name because to ensure that cell names are unique, KLayout may assign different names to the actual cell compared to the source cell.

This method has been introduced in version 0.22.

bbox

(1) Signature: *[const]* Box **bbox**

Description: Gets the bounding box of the cell

Returns: The bounding box of the cell

The bounding box is computed over all layers. To compute the bounding box over single layers, use bbox with a layer index argument.

(2) Signature: *[const]* Box **bbox** (unsigned int layer_index)

Description: Gets the per-layer bounding box of the cell

Returns: The bounding box of the cell considering only the given layer

The bounding box is the box enclosing all shapes on the given layer.

'bbox' is the preferred synonym since version 0.28.

bbox_per_layer

Signature: *[const]* Box **bbox_per_layer** (unsigned int layer_index)

Description: Gets the per-layer bounding box of the cell

Returns: The bounding box of the cell considering only the given layer

Use of this method is deprecated. Use bbox instead

The bounding box is the box enclosing all shapes on the given layer.

'bbox' is the preferred synonym since version 0.28.

begin_instances_rec

Signature: *[const]* RecursiveInstanceIterator **begin_instances_rec**

Description: Delivers a recursive instance iterator for the instances below the cell

Returns: A suitable iterator

For details see the description of the RecursiveInstanceIterator class.

This method has been added in version 0.27.

begin_instances_rec_overlapping

(1) Signature: *[const]* RecursiveInstanceIterator **begin_instances_rec_overlapping** (Box region)

Description: Delivers a recursive instance iterator for the instances below the cell using a region search

region: The search region

Returns: A suitable iterator

For details see the description of the RecursiveInstanceIterator class. This version gives an iterator delivering instances whose bounding box overlaps the given region.

This method has been added in version 0.27.

(2) Signature: *[const]* RecursiveInstanceIterator **begin_instances_rec_overlapping** (DBox region)

Description: Delivers a recursive instance iterator for the instances below the cell using a region search, with the region given in micrometer units

region: The search region as DBox object in micrometer units

Returns: A suitable iterator

For details see the description of the RecursiveInstanceIterator class. This version gives an iterator delivering instances whose bounding box overlaps the given region.

This variant has been added in version 0.27.

begin_instances_rec_touching

(1) Signature: *[const]* RecursiveInstanceIterator **begin_instances_rec_touching** (Box region)

Description: Delivers a recursive instance iterator for the instances below the cell

region: The search region

Returns: A suitable iterator

For details see the description of the RecursiveInstanceIterator class. This version gives an iterator delivering instances whose bounding box touches the given region.

This method has been added in version 0.27.

(2) Signature: *[const]* RecursiveInstanceIterator **begin_instances_rec_touching** (DBox region)

Description: Delivers a recursive instance iterator for the instances below the cell using a region search, with the region given in micrometer units

region: The search region as DBox object in micrometer units

Returns: A suitable iterator

For details see the description of the RecursiveInstanceIterator class. This version gives an iterator delivering instances whose bounding box touches the given region.

This variant has been added in version 0.27.

begin_shapes_rec

Signature: *[const]* RecursiveShapeIterator **begin_shapes_rec** (unsigned int layer)

Description: Delivers a recursive shape iterator for the shapes below the cell on the given layer

layer:	The layer from which to get the shapes
Returns:	A suitable iterator

For details see the description of the RecursiveShapeIterator class.

This method has been added in version 0.23.

begin_shapes_rec_overlapping

(1) Signature: *[const]* RecursiveShapeIterator **begin_shapes_rec_overlapping** (unsigned int layer, Box region)

Description: Delivers a recursive shape iterator for the shapes below the cell on the given layer using a region search

layer:	The layer from which to get the shapes
region:	The search region
Returns:	A suitable iterator

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box overlaps the given region.

This method has been added in version 0.23.

(2) Signature: *[const]* RecursiveShapeIterator **begin_shapes_rec_overlapping** (unsigned int layer, DBox region)

Description: Delivers a recursive shape iterator for the shapes below the cell on the given layer using a region search, with the region given in micrometer units

layer:	The layer from which to get the shapes
region:	The search region as DBox object in micrometer units
Returns:	A suitable iterator

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box overlaps the given region.

This variant has been added in version 0.25.

begin_shapes_rec_touching

(1) Signature: *[const]* RecursiveShapeIterator **begin_shapes_rec_touching** (unsigned int layer, Box region)

Description: Delivers a recursive shape iterator for the shapes below the cell on the given layer using a region search

layer:	The layer from which to get the shapes
region:	The search region
Returns:	A suitable iterator

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box touches the given region.

This method has been added in version 0.23.

(2) Signature: *[const]* RecursiveShapeIterator **begin_shapes_rec_touching** (unsigned int layer, DBox region)

Description: Delivers a recursive shape iterator for the shapes below the cell on the given layer using a region search, with the region given in micrometer units

layer:	The layer from which to get the shapes

region: The search region as DBox object in micrometer units

Returns: A suitable iterator

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box touches the given region.

This variant has been added in version 0.25.

called_cells

Signature: *[const]* unsigned int[] **called_cells**

Description: Gets a list of all called cells

Returns: A list of cell indices.

This method determines all cells which are called either directly or indirectly by the cell. It returns an array of cell indexes. Use the 'cell' method of Layout to retrieve the corresponding Cell object.

This method has been introduced in version 0.19.

caller_cells

Signature: *[const]* unsigned int[] **caller_cells**

Description: Gets a list of all caller cells

Returns: A list of cell indices.

This method determines all cells which call this cell either directly or indirectly. It returns an array of cell indexes. Use the 'cell' method of Layout to retrieve the corresponding Cell object.

This method has been introduced in version 0.19.

cell_index

Signature: *[const]* unsigned int **cell_index**

Description: Gets the cell index

Returns: The cell index of the cell

change_pcell_parameter

Signature: Instance **change_pcell_parameter** (const Instance instance, string name, variant value)

Description: Changes a single parameter for an individual PCell instance given by name

Returns: The new instance (the old may be invalid)

This will set the PCell parameter named 'name' to the given value for the instance addressed by 'instance'. If no parameter with that name exists, the method will do nothing.

This method has been introduced in version 0.23.

change_pcell_parameters

(1) Signature: Instance **change_pcell_parameters** (const Instance instance, map<string,variant> dict)

Description: Changes the given parameter for an individual PCell instance

Returns: The new instance (the old may be invalid)

This version receives a dictionary of names and values. It will change the parameters given by the names to the values given by the values of the dictionary. The functionality is similar to the same function with an array, but more convenient to use. Values with unknown names are ignored.

This method has been introduced in version 0.24.

(2) Signature: Instance **change_pcell_parameters** (const Instance instance, variant[] parameters)

Description: Changes the parameters for an individual PCell instance

Returns: The new instance (the old may be invalid)

If necessary, this method creates a new variant and replaces the given instance by an instance of this variant.

The parameters are given in the order the parameters are declared. Use pcell_declaration on the instance to get the PCell declaration object of the cell. That PCellDeclaration object delivers the parameter declaration with its 'get_parameters' method. Each parameter in the variant list passed to the second list of values corresponds to one parameter declaration.

There is a more convenient method (change_pcell_parameter) that changes a single parameter by name.

This method has been introduced in version 0.22.

child_cells

Signature: *[const]* unsigned long **child_cells**

Description: Gets the number of child cells

The number of child cells (not child instances!) is returned. CAUTION: this method is SLOW, in particular if many instances are present.

child_instances

Signature: *[const]* unsigned long **child_instances**

Description: Gets the number of child instances

Returns: Returns the number of cell instances

clear

(1) Signature: void **clear** (unsigned int layer_index)

Description: Clears the shapes on the given layer

(2) Signature: void **clear**

Description: Clears the cell (deletes shapes and instances)

This method has been introduced in version 0.23.

clear_insts

Signature: void **clear_insts**

Description: Clears the instance list

clear_meta_info

Signature: void **clear_meta_info**

Description: Clears the meta information of the cell

See LayoutMetaInfo for details about cells and meta information.

This method has been introduced in version 0.28.8.

clear_shapes

Signature: void **clear_shapes**

Description: Clears all shapes in the cell

copy

(1) Signature: void **copy** (unsigned int src, unsigned int dest)

Description: Copies the shapes from the source to the target layer

src: The layer index of the source layer

dest: The layer index of the destination layer

The destination layer is not overwritten. Instead, the shapes are added to the shapes of the destination layer. If source are target layer are identical, this method does nothing. This method will copy shapes within the cell. To copy shapes from another cell to this cell, use the copy method with the cell parameter.

This method has been introduced in version 0.19.

(2) Signature: void **copy** (Cell ptr src_cell, unsigned int src_layer, unsigned int dest)

Description: Copies shapes from another cell to the target layer in this cell

src_cell:	The cell where to take the shapes from
src_layer:	The layer index of the layer from which to take the shapes
dest:	The layer index of the destination layer

This method will copy all shapes on layer 'src_layer' of cell 'src_cell' to the layer 'dest' of this cell. The destination layer is not overwritten. Instead, the shapes are added to the shapes of the destination layer. If the source cell lives in a layout with a different database unit than that current cell is in, the shapes will be transformed accordingly. The same way, shape properties are transformed as well. Note that the shape transformation may require rounding to smaller coordinates. This may result in a slight distortion of the original shapes, in particular when transforming into a layout with a bigger database unit.

copy_instances

Signature: void **copy_instances** (const Cell source_cell)

Description: Copies the instances of child cells in the source cell to this cell

source_cell:	The cell where the instances are copied from

The source cell must reside in the same layout than this cell. The instances of child cells inside the source cell are copied to this cell. No new cells are created, just new instances are created to already existing cells in the target cell.

The instances will be added to any existing instances in the cell.

More elaborate methods of copying hierarchy trees between layouts or duplicating trees are provided through the copy_tree_shapes (in cooperation with the CellMapping class) or copy_tree methods.

This method has been added in version 0.23.

copy_meta_info

Signature: void **copy_meta_info** (const Cell ptr other)

Description: Copies the meta information from the other cell into this cell

See LayoutMetaInfo for details about cells and meta information. The meta information from this cell will be replaced by the meta information from the other cell.

This method has been introduced in version 0.28.16.

copy_shapes

(1) Signature: void **copy_shapes** (const Cell source_cell)

Description: Copies the shapes from the given cell into this cell

source_cell:	The cell from where to copy shapes

All shapes are copied from the source cell to this cell. Instances are not copied.

The source cell can reside in a different layout. In this case, the shapes are copied over from the other layout into this layout. Database unit conversion is done automatically if the database units differ between the layouts. Note that this may lead to grid snapping effects if the database unit of the target layout is not an integer fraction of the source layout.

If source and target layout are different, the layers of the source and target layout are identified by their layer/datatype number or name (if no layer/datatype is present). The shapes will be added to any shapes already in the cell.

This method has been added in version 0.23.

(2) Signature: void **copy_shapes** (const Cell source_cell, const LayerMapping layer_mapping)

Description: Copies the shapes from the given cell into this cell

source_cell: The cell from where to copy shapes

layer_mapping: A LayerMapping object that specifies which layers are copied and where

All shapes on layers specified in the layer mapping object are copied from the source cell to this cell. Instances are not copied. The target layer is taken from the mapping table.

The shapes will be added to any shapes already in the cell.

This method has been added in version 0.23.

copy_tree

Signature: unsigned int[] **copy_tree** (const Cell source_cell)

Description: Copies the cell tree of the given cell into this cell

source_cell: The cell from where to copy the cell tree

Returns: A list of indexes of newly created cells

The complete cell tree of the source cell is copied to the target cell plus all shapes in that tree are copied as well. This method will basically duplicate the cell tree of the source cell.

The source cell may reside in a separate layout. This method therefore provides a way to copy over complete cell trees from one layout to another.

The shapes and instances will be added to any shapes or instances already in the cell.

This method has been added in version 0.23.

copy_tree_shapes

(1) Signature: void **copy_tree_shapes** (const Cell source_cell, const CellMapping cell_mapping)

Description: Copies the shapes from the given cell and the cell tree below into this cell or subcells of this cell

source_cell: The starting cell from where to copy shapes

cell_mapping: The cell mapping object that determines how cells are identified between source and target layout

This method is provided if source and target cell reside in different layouts. If will copy the shapes from all cells below the given source cell, but use a cell mapping object that provides a specification how cells are identified between the layouts. Cells in the source tree, for which no mapping is provided, will be flattened - their shapes will be propagated into parent cells for which a mapping is provided.

The cell mapping object provides various methods to map cell trees between layouts. See the CellMapping class for details about the mapping methods available. The cell mapping object is also responsible for creating a proper hierarchy of cells in the target layout if that is required.

Layers are identified between the layouts by the layer/datatype number of name if no layer/datatype number is present.

The shapes copied will be added to any shapes already in the cells.

This method has been added in version 0.23.

(2) Signature: void **copy_tree_shapes** (const Cell source_cell, const CellMapping cell_mapping, const LayerMapping layer_mapping)

Description: Copies the shapes from the given cell and the cell tree below into this cell or subcells of this cell with layer mapping

source_cell: The cell from where to copy shapes and instances

cell_mapping: The cell mapping object that determines how cells are identified between source and target layout

This method is provided if source and target cell reside in different layouts. If will copy the shapes from all cells below the given source cell, but use a cell mapping object that provides a specification how cells are identified between the layouts. Cells in the source tree, for which no mapping is

provided, will be flattened - their shapes will be propagated into parent cells for which a mapping is provided.

The cell mapping object provides various methods to map cell trees between layouts. See the CellMapping class for details about the mapping methods available. The cell mapping object is also responsible for creating a proper hierarchy of cells in the target layout if that is required.

In addition, the layer mapping object can be specified which maps source to target layers. This feature can be used to restrict the copy operation to a subset of layers or to convert shapes to different layers in that step.

The shapes copied will be added to any shapes already in the cells.

This method has been added in version 0.23.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

dbbox

(1) Signature: *[const]* DBox **dbbox**

Description: Gets the bounding box of the cell in micrometer units

Returns: The bounding box of the cell

The bounding box is computed over all layers. To compute the bounding box over single layers, use dbbox with a layer index argument.

This method has been introduced in version 0.25.

(2) Signature: *[const]* DBox **dbbox** (unsigned int layer_index)

Description: Gets the per-layer bounding box of the cell in micrometer units

Returns: The bounding box of the cell considering only the given layer

The bounding box is the box enclosing all shapes on the given layer.

This method has been introduced in version 0.25. 'dbbox' is the preferred synonym since version 0.28.

dbbox_per_layer

Signature: *[const]* DBox **dbbox_per_layer** (unsigned int layer_index)

Description: Gets the per-layer bounding box of the cell in micrometer units

Returns: The bounding box of the cell considering only the given layer

Use of this method is deprecated. Use dbbox instead

The bounding box is the box enclosing all shapes on the given layer.

This method has been introduced in version 0.25. 'dbbox' is the preferred synonym since version 0.28.

delete

Signature: void **delete**

Description: Deletes this cell

This deletes the cell but not the sub cells of the cell. These subcells will likely become new top cells unless they are used otherwise. All instances of this cell are deleted as well. Hint: to delete multiple cells, use "delete_cells" which is far more efficient in this case.

After the cell has been deleted, the Cell object becomes invalid. Do not access methods or attributes of this object after deleting the cell.

This method has been introduced in version 0.23.

delete_property

Signature: void **delete_property** (variant key)

Description: Deletes the user property with the given key

This method is a convenience method that deletes the property with the given key. It does nothing if no property with that key exists. Using that method is more convenient than creating a new property set with a new ID and assigning that properties ID. This method may change the properties ID.

This method has been introduced in version 0.23.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

display_title

Signature: *[const]* string **display_title**

Description: Returns a nice looking name for display purposes

For example, this name include PCell parameters for PCell proxy cells.

This method has been introduced in version 0.22.

dup

Signature: *[const]* Cell ptr **dup**

Description: Creates a copy of the cell

This method will create a copy of the cell. The new cell will be member of the same layout the original cell was member of. The copy will inherit all shapes and instances, but get a different cell_index and a modified name as duplicate cell names are not allowed in the same layout.

This method has been introduced in version 0.27.

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_child_cell

Signature: *[const,iter]* unsigned int **each_child_cell**

Description: Iterates over all child cells

This iterator will report the child cell indices, not every instance.

each_inst

Signature: *[iter]* Instance **each_inst**

Description: Iterates over all child instances (which may actually be instance arrays)

Starting with version 0.15, this iterator delivers Instance objects rather than CellInstArray objects.

each_meta_info

Signature: *[const,iter]* LayoutMetaInfo **each_meta_info**

Description: Iterates over the meta information of the cell

See LayoutMetaInfo for details about cells and meta information.

This method has been introduced in version 0.28.8.

each_overlapping_inst

(1) Signature: *[const,iter]* Instance **each_overlapping_inst** (const Box b)

Description: Gets the instances overlapping the given rectangle

b: The region to iterate over

This will iterate over all child cell instances overlapping with the given rectangle b.

Starting with version 0.15, this iterator delivers Instance objects rather than CellInstArray objects.

(2) Signature: *[const,iter]* Instance **each_overlapping_inst** (const DBox b)

Description: Gets the instances overlapping the given rectangle, with the rectangle in micrometer units

b: The region to iterate over

This will iterate over all child cell instances overlapping with the given rectangle b. This method is identical to the each_overlapping_inst version that takes a Box object, but instead of taking database unit coordinates in will take a micrometer unit DBox object.

This variant has been introduced in version 0.25.

each_overlapping_shape

(1) Signature: *[const,iter]* Shape **each_overlapping_shape** (unsigned int layer_index, const Box box, unsigned int flags)

Description: Iterates over all shapes of a given layer that overlap the given box

flags: An "or"-ed combination of the S.. constants of the Shapes class

box: The box by which to query the shapes

layer_index: The layer on which to run the query

(2) Signature: *[const,iter]* Shape **each_overlapping_shape** (unsigned int layer_index, const Box box)

Description: Iterates over all shapes of a given layer that overlap the given box

box: The box by which to query the shapes

layer_index: The layer on which to run the query

This call is equivalent to each_overlapping_shape(layer_index,box,RBA::Shapes::SAll). This convenience method has been introduced in version 0.16.

(3) Signature: *[const,iter]* Shape **each_overlapping_shape** (unsigned int layer_index, const DBox box, unsigned int flags)

Description: Iterates over all shapes of a given layer that overlap the given box, with the box given in micrometer units

flags: An "or"-ed combination of the S.. constants of the Shapes class

box: The box by which to query the shapes as a DBox object in micrometer units

layer_index: The layer on which to run the query

(4) Signature: *[const,iter]* Shape **each_overlapping_shape** (unsigned int layer_index, const DBox box)

Description: Iterates over all shapes of a given layer that overlap the given box, with the box given in micrometer units

box: The box by which to query the shapes as a DBox object in micrometer units

layer_index: The layer on which to run the query

This call is equivalent to each_overlapping_shape(layer_index,box,RBA::Shapes::SAll). This convenience method has been introduced in version 0.16.

each_parent_cell

Signature: *[const,iter]* unsigned int **each_parent_cell**

Description: Iterates over all parent cells

This iterator will iterate over the parent cells, just returning their cell index.

each_parent_inst

Signature: *[const,iter]* ParentInstArray **each_parent_inst**

Description: Iterates over the parent instance list (which may actually be instance arrays)

The parent instances are basically inversions of the instances. Using parent instances it is possible to determine how a specific cell is called from where.

each_shape

(1) Signature: *[const,iter]* Shape **each_shape** (unsigned int layer_index, unsigned int flags)

Description: Iterates over all shapes of a given layer

flags: An "or"-ed combination of the S.. constants of the Shapes class

layer_index: The layer on which to run the query

This iterator is equivalent to 'shapes(layer).each'.

(2) Signature: *[const,iter]* Shape **each_shape** (unsigned int layer_index)

Description: Iterates over all shapes of a given layer

layer_index: The layer on which to run the query

This call is equivalent to each_shape(layer_index,RBA::Shapes::SAll). This convenience method has been introduced in version 0.16.

each_touching_inst

(1) Signature: *[const,iter]* Instance **each_touching_inst** (const Box b)

Description: Gets the instances touching the given rectangle

b: The region to iterate over

This will iterate over all child cell instances overlapping with the given rectangle b.

Starting with version 0.15, this iterator delivers Instance objects rather than CellInstArray objects.

(2) Signature: *[const,iter]* Instance **each_touching_inst** (const DBox b)

Description: Gets the instances touching the given rectangle, with the rectangle in micrometer units

b: The region to iterate over

This will iterate over all child cell instances touching the given rectangle b. This method is identical to the each_touching_inst version that takes a Box object, but instead of taking database unit coordinates in will take a micrometer unit DBox object.

This variant has been introduced in version 0.25.

each_touching_shape

(1) Signature: *[const,iter]* Shape **each_touching_shape** (unsigned int layer_index, const Box box, unsigned int flags)

Description: Iterates over all shapes of a given layer that touch the given box

flags:	An "or"-ed combination of the S.. constants of the Shapes class
box:	The box by which to query the shapes
layer_index:	The layer on which to run the query

(2) Signature: *[const,iter]* Shape **each_touching_shape** (unsigned int layer_index, const Box box)

Description: Iterates over all shapes of a given layer that touch the given box

box:	The box by which to query the shapes
layer_index:	The layer on which to run the query

This call is equivalent to each_touching_shape(layer_index,box,RBA::Shapes::SAll). This convenience method has been introduced in version 0.16.

(3) Signature: *[const,iter]* Shape **each_touching_shape** (unsigned int layer_index, const DBox box, unsigned int flags)

Description: Iterates over all shapes of a given layer that touch the given box, with the box given in micrometer units

flags:	An "or"-ed combination of the S.. constants of the Shapes class
box:	The box by which to query the shapes as a DBox object in micrometer units
layer_index:	The layer on which to run the query

(4) Signature: *[const,iter]* Shape **each_touching_shape** (unsigned int layer_index, const DBox box)

Description: Iterates over all shapes of a given layer that touch the given box, with the box given in micrometer units

box:	The box by which to query the shapes as a DBox object in micrometer units
layer_index:	The layer on which to run the query

This call is equivalent to each_touching_shape(layer_index,box,RBA::Shapes::SAll). This convenience method has been introduced in version 0.16.

erase

Signature: void **erase** (const Instance inst)

Description: Erases the instance given by the Instance object

This method has been introduced in version 0.16. It can only be used in editable mode.

fill_region

(1) Signature: void **fill_region** (const Region region, unsigned int fill_cell_index, const Box fc_box, const Point ptr origin = (0, 0), Region ptr remaining_parts = nil, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil, const Box glue_box = ())

Description: Fills the given region with cells of the given type (extended version)

region:	The region to fill
fill_cell_index:	The fill cell to place
fc_box:	The fill cell's footprint
origin:	The global origin of the fill pattern or nil to allow local (per-polygon) optimization
remaining_parts:	See explanation below
fill_margin:	See explanation below
remaining_polygons:	See explanation below
glue_box:	Guarantees fill cell compatibility to neighbor regions in enhanced mode

This method creates a regular pattern of fill cells to cover the interior of the given region as far as possible. This process is also known as tiling. This implementation supports rectangular (not necessarily square) tile cells. The tile cell's footprint is given by the fc_box parameter and the cells will be arranged with their footprints forming a seamless array.

The algorithm supports a global fill raster as well as local (per-polygon) origin optimization. In the latter case the origin of the regular raster is optimized per individual polygon of the fill region. To enable optimization, pass 'nil' to the 'origin' argument.

The implementation will basically try to find a repetition pattern of the tile cell's footprint and produce instances which fit entirely into the fill region.

There is also a version available which offers skew step vectors as a generalization of the orthogonal ones.

If the 'remaining_parts' argument is non-nil, the corresponding region will receive the parts of the polygons which are not covered by tiles. Basically the tiles are subtracted from the original polygons. A margin can be specified which is applied separately in x and y direction before the subtraction is done ('fill_margin' parameter).

If the 'remaining_polygons' argument is non-nil, the corresponding region will receive all polygons from the input region which could not be filled and where there is no chance of filling because not a single tile will fit into them.

'remaining_parts' and 'remaining_polygons' can be identical with the input. In that case the input will be overwritten with the respective output. Otherwise, the respective polygons are added to these regions.

This allows setting up a more elaborate fill scheme using multiple iterations and local origin-optimization ('origin' is nil):

```
r = ...        # region to fill
c = ...        # cell in which to produce the fill cells
fc_index = ... # fill cell index
fc_box = ...   # fill cell footprint

fill_margin = RBA::Point::new(0, 0)   # x/y distance between tile cells with
 different origin

# Iteration: fill a region and fill the remaining parts as long as there is
 anything left.
# Polygons not worth being considered further are dropped (last argument is
 nil).
while !r.is_empty?
  c.fill_region(r, fc_index, fc_box, nil, r, fill_margin, nil)
```

```
end
```

The glue box parameter supports fill cell array compatibility with neighboring regions. This is specifically useful when putting the fill_cell method into a tiling processor. Fill cell array compatibility means that the fill cell array continues over tile boundaries. This is easy with an origin: you can chose the origin identically over all tiles which is sufficient to guarantee fill cell array compatibility across the tiles. However there is no freedom of choice of the origin then and fill cell placement may not be optimal. To enable the origin for the tile boundary only, a glue box can given. The origin will then be used only when the polygons to fill not entirely inside and not at the border of the glue box. Hence, while a certain degree of freedom is present for the placement of fill cells inside the glue box, the fill cells are guaranteed to be placed at the raster implied by origin at the glue box border and beyond. To ensure fill cell compatibility inside the tiling processor, it is sufficient to use the tile box as the glue box.

This method has been introduced in version 0.23 and enhanced in version 0.27.

(2) Signature: void **fill_region** (const Region region, unsigned int fill_cell_index, const Box fc_bbox, const Vector row_step, const Vector column_step, const Point ptr origin = (0, 0), Region ptr remaining_parts = nil, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil, const Box glue_box = ())

Description: Fills the given region with cells of the given type (skew step version)

region:	The region to fill
fill_cell_index:	The fill cell to place
fc_bbox:	The fill cell's box to place
row_step:	The 'rows' step vector
column_step:	The 'columns' step vector
origin:	The global origin of the fill pattern or nil to allow local (per-polygon) optimization
remaining_parts:	See explanation in other version
fill_margin:	See explanation in other version
remaining_polygons:	See explanation in other version

This version is similar to the version providing an orthogonal fill, but it offers more generic stepping of the fill cell. The step pattern is defined by an origin and two vectors (row_step and column_step) which span the axes of the fill cell pattern.

The fill box and the step vectors are decoupled which means the fill box can be larger or smaller than the step pitch - it can be overlapping and there can be space between the fill box instances. Fill boxes are placed where they fit entirely into a polygon of the region. The fill boxes lower left corner is the reference for the fill pattern and aligns with the origin if given.

This variant has been introduced in version 0.27.

fill_region_multi

Signature: void **fill_region_multi** (const Region region, unsigned int fill_cell_index, const Box fc_bbox, const Vector row_step, const Vector column_step, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil, const Box glue_box = ())

Description: Fills the given region with cells of the given type in enhanced mode with iterations

This version operates like fill_region, but repeats the fill generation until no further fill cells can be placed. As the fill pattern origin changes between the iterations, narrow regions can be filled which cannot with a fixed fill pattern origin. The fill_margin parameter is important as it controls the distance between fill cells with a different origin and therefore introduces a safety distance between pitch-incompatible arrays.

The origin is ignored unless a glue box is given. See fill_region for a description of this concept.

This method has been introduced in version 0.27.

flatten

(1) Signature: void **flatten** (bool prune)

Description: Flattens the given cell

prune: Set to true to remove orphan cells.

This method propagates all shapes from the hierarchy below into the given cell. It also removes the instances of the cells from which the shapes came from, but does not remove the cells themselves if prune is set to false. If prune is set to true, these cells are removed if not used otherwise.

A version of this method exists which allows one to specify the number of hierarchy levels to which subcells are considered.

This method has been introduced in version 0.23.

(2) Signature: void **flatten** (int levels, bool prune)

Description: Flattens the given cell

levels: The number of hierarchy levels to flatten (-1: all, 0: none, 1: one level etc.)

prune: Set to true to remove orphan cells.

This method propagates all shapes from the specified number of hierarchy levels below into the given cell. It also removes the instances of the cells from which the shapes came from, but does not remove the cells themselves if prune is set to false. If prune is set to true, these cells are removed if not used otherwise.

This method has been introduced in version 0.23.

ghost_cell=

Signature: void **ghost_cell=** (bool flag)

Description: Sets the "ghost cell" flag

See is_ghost_cell? for a description of this property.

This method has been introduced in version 0.20.

Python specific notes:
The object exposes a writable attribute 'ghost_cell'. This is the setter.

has_prop_id?

Signature: *[const]* bool **has_prop_id?**

Description: Returns true, if the cell has user properties

This method has been introduced in version 0.23.

hierarchy_levels

Signature: *[const]* unsigned int **hierarchy_levels**

Description: Returns the number of hierarchy levels below

This method returns the number of call levels below the current cell. If there are no child cells, this method will return 0, if there are only direct children, it will return 1.

CAUTION: this method may be expensive!

insert

(1) Signature: Instance **insert** (const Instance inst)

Description: Inserts a cell instance given by another reference

Returns: An Instance object representing the new instance

This method allows one to copy instances taken from a reference (an Instance object). This method is not suited to inserting instances from other Layouts into this cell. For this purpose, the hierarchical copy methods of Layout have to be used.

It has been added in version 0.16.

(2) Signature: Instance **insert** (const CellInstArray cell_inst_array)

Description: Inserts a cell instance (array)

Returns: An Instance object representing the new instance

With version 0.16, this method returns an Instance object that represents the new instance. It's use is discouraged in readonly mode, since it invalidates other Instance references.

(3) Signature: Instance **insert** (const DCellInstArray cell_inst_array)

Description: Inserts a cell instance (array) given in micron units

Returns: An Instance object representing the new instance

This method inserts an instance array, similar to insert with a CellInstArray parameter. But in this version, the argument is a cell instance array given in micrometer units. It is translated to database units internally.

This variant has been introduced in version 0.25.

(4) Signature: Instance **insert** (const DCellInstArray cell_inst_array, unsigned long property_id)

Description: Inserts a cell instance (array) given in micron units with properties

Returns: An Instance object representing the new instance

This method inserts an instance array, similar to insert with a CellInstArray parameter and a property set ID. But in this version, the argument is a cell instance array given in micrometer units. It is translated to database units internally.

This variant has been introduced in version 0.25.

(5) Signature: Instance **insert** (const CellInstArray cell_inst_array, unsigned long property_id)

Description: Inserts a cell instance (array) with properties

Returns: An Instance object representing the new instance

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. With version 0.16, this method returns an Instance object that represents the new instance. It's use is discouraged in readonly mode, since it invalidates other Instance references.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_empty?

Signature: *[const]* bool **is_empty?**

Description: Returns a value indicating whether the cell is empty

An empty cell is a cell not containing instances nor any shapes.

This method has been introduced in version 0.20.

is_ghost_cell?

Signature: *[const]* bool **is_ghost_cell?**

Description: Returns a value indicating whether the cell is a "ghost cell"

The ghost cell flag is used by the GDS reader for example to indicate that the cell is not located inside the file. Upon writing the reader can determine whether to write the cell or not. To satisfy the references inside the layout, a dummy cell is created in this case which has the "ghost cell" flag set to true.

This method has been introduced in version 0.20.

Python specific notes:
The object exposes a readable attribute 'ghost_cell'. This is the getter.

is_leaf?

Signature: *[const]* bool **is_leaf?**

Description: Gets a value indicating whether the cell is a leaf cell

A cell is a leaf cell if there are no child instantiations.

is_library_cell?

Signature: *[const]* bool **is_library_cell?**

Description: Returns true, if the cell is a proxy cell pointing to a library cell

If the cell is imported from some library, this attribute returns true. Please note, that this attribute can combine with is_pcell? for PCells imported from a library.

This method has been introduced in version 0.22.

is_pcell_variant?

(1) Signature: *[const]* bool **is_pcell_variant?**

Description: Returns true, if this cell is a pcell variant

this method returns true, if this cell represents a pcell with a distinct set of parameters (a PCell proxy). This also is true, if the PCell is imported from a library.

Technically, PCells imported from a library are library proxies which are pointing to PCell variant proxies. This scheme can even proceed over multiple indirections, i.e. a library using PCells from another library.

This method has been introduced in version 0.22.

(2) Signature: *[const]* bool **is_pcell_variant?** (const Instance instance)

Description: Returns true, if this instance is a PCell variant

This method returns true, if this instance represents a PCell with a distinct set of parameters. This method also returns true, if it is a PCell imported from a library.

This method has been introduced in version 0.22.

is_proxy?

Signature: *[const]* bool **is_proxy?**

Description: Returns true, if the cell presents some external entity

A cell may represent some data which is imported from some other source, i.e. a library. Such cells are called "proxy cells". For a library reference, the proxy cell is some kind of pointer to the library and the cell within the library.

For PCells, this data can even be computed through some script. A PCell proxy represents all instances with a given set of parameters.

Proxy cells cannot be modified, except that pcell parameters can be modified and PCell instances can be recomputed.

This method has been introduced in version 0.22.

is_top?

Signature: *[const]* bool **is_top?**

Description: Gets a value indicating whether the cell is a top-level cell

A cell is a top-level cell if there are no parent instantiations.

is_valid?

Signature: *[const]* bool **is_valid?** (const Instance instance)

Description: Tests if the given Instance object is still pointing to a valid object

This method has been introduced in version 0.16. If the instance represented by the given reference has been deleted, this method returns false. If however, another instance has been inserted already that occupies the original instances position, this method will return true again.

layout

(1) Signature: Layout ptr **layout**

Description: Returns a reference to the layout where the cell resides

this method has been introduced in version 0.22.

(2) Signature: *[const]* const Layout ptr **layout**

Description: Returns a reference to the layout where the cell resides (const references)

this method has been introduced in version 0.22.

library

Signature: *[const]* Library ptr **library**

Description: Returns a reference to the library from which the cell is imported

if the cell is not imported from a library, this reference is nil.

this method has been introduced in version 0.22.

library_cell_index

Signature: *[const]* unsigned int **library_cell_index**

Description: Returns the index of the cell in the layout of the library (if it's a library proxy)

Together with the library method, it is possible to locate the source cell of a library proxy. The source cell can be retrieved from a cell "c" with

```
c.library.layout.cell(c.library_cell_index)
```

This cell may be itself a proxy, i.e. for pcell libraries, where the library cells are pcell variants which itself are proxies to a pcell.

This method has been introduced in version 0.22.

merge_meta_info

Signature: void **merge_meta_info** (const Cell ptr other)

Description: Merges the meta information from the other cell into this cell

See LayoutMetaInfo for details about cells and meta information. Existing keys in this cell will be overwritten by the respective values from the other cell. New keys will be added.

This method has been introduced in version 0.28.16.

meta_info

Signature: new LayoutMetaInfo ptr **meta_info** (string name)

Description: Gets the meta information for a given name

See LayoutMetaInfo for details about cells and meta information.

If no meta information with the given name exists, a default object with empty fields will be returned.

This method has been introduced in version 0.28.8.

meta_info_value

Signature: variant **meta_info_value** (string name)

Description: Gets the meta information value for a given name

See LayoutMetaInfo for details about cells and meta information.

If no meta information with the given name exists, a nil value will be returned. A more generic version that delivers all fields of the meta information is meta_info.

This method has been introduced in version 0.28.8.

move

(1) Signature: void **move** (unsigned int src, unsigned int dest)

Description: Moves the shapes from the source to the target layer

src:	The layer index of the source layer
dest:	The layer index of the destination layer

The destination layer is not overwritten. Instead, the shapes are added to the shapes of the destination layer. This method will move shapes within the cell. To move shapes from another cell to this cell, use the copy method with the cell parameter.

This method has been introduced in version 0.19.

(2) Signature: void **move** (Cell ptr src_cell, unsigned int src_layer, unsigned int dest)

Description: Moves shapes from another cell to the target layer in this cell

src_cell:	The cell where to take the shapes from
src_layer:	The layer index of the layer from which to take the shapes
dest:	The layer index of the destination layer

This method will move all shapes on layer 'src_layer' of cell 'src_cell' to the layer 'dest' of this cell. The destination layer is not overwritten. Instead, the shapes are added to the shapes of the destination layer. If the source cell lives in a layout with a different database unit than that current cell is in, the shapes will be transformed accordingly. The same way, shape properties are transformed as well. Note that the shape transformation may require rounding to smaller coordinates. This may result in a slight distortion of the original shapes, in particular when transforming into a layout with a bigger database unit.

move_instances

Signature: void **move_instances** (Cell source_cell)

Description: Moves the instances of child cells in the source cell to this cell

source_cell:	The cell where the instances are moved from

The source cell must reside in the same layout than this cell. The instances of child cells inside the source cell are moved to this cell. No new cells are created, just new instances are created to already existing cells in the target cell.

The instances will be added to any existing instances in the cell.

More elaborate methods of moving hierarchy trees between layouts are provided through the move_tree_shapes (in cooperation with the CellMapping class) or move_tree methods.

This method has been added in version 0.23.

move_shapes

(1) Signature: void **move_shapes** (Cell source_cell)

Description: Moves the shapes from the given cell into this cell

source_cell:	The cell from where to move shapes

All shapes are moved from the source cell to this cell. Instances are not moved.

The source cell can reside in a different layout. In this case, the shapes are moved over from the other layout into this layout. Database unit conversion is done automatically if the database units

differ between the layouts. Note that this may lead to grid snapping effects if the database unit of the target layout is not an integer fraction of the source layout.

If source and target layout are different, the layers of the source and target layout are identified by their layer/datatype number or name (if no layer/datatype is present). The shapes will be added to any shapes already in the cell.

This method has been added in version 0.23.

(2) Signature: void **move_shapes** (Cell source_cell, const LayerMapping layer_mapping)

Description: Moves the shapes from the given cell into this cell

source_cell:	The cell from where to move shapes
layer_mapping:	A LayerMapping object that specifies which layers are moved and where

All shapes on layers specified in the layer mapping object are moved from the source cell to this cell. Instances are not moved. The target layer is taken from the mapping table.

The shapes will be added to any shapes already in the cell.

This method has been added in version 0.23.

move_tree

Signature: unsigned int[] **move_tree** (Cell source_cell)

Description: Moves the cell tree of the given cell into this cell

source_cell:	The cell from where to move the cell tree
Returns:	A list of indexes of newly created cells

The complete cell tree of the source cell is moved to the target cell plus all shapes in that tree are moved as well. This method will basically rebuild the cell tree of the source cell and empty the source cell.

The source cell may reside in a separate layout. This method therefore provides a way to move over complete cell trees from one layout to another.

The shapes and instances will be added to any shapes or instances already in the cell.

This method has been added in version 0.23.

move_tree_shapes

(1) Signature: void **move_tree_shapes** (Cell source_cell, const CellMapping cell_mapping)

Description: Moves the shapes from the given cell and the cell tree below into this cell or subcells of this cell

source_cell:	The starting cell from where to move shapes
cell_mapping:	The cell mapping object that determines how cells are identified between source and target layout

This method is provided if source and target cell reside in different layouts. If will move the shapes from all cells below the given source cell, but use a cell mapping object that provides a specification how cells are identified between the layouts. Cells in the source tree, for which no mapping is provided, will be flattened - their shapes will be propagated into parent cells for which a mapping is provided.

The cell mapping object provides various methods to map cell trees between layouts. See the CellMapping class for details about the mapping methods available. The cell mapping object is also responsible for creating a proper hierarchy of cells in the target layout if that is required.

Layers are identified between the layouts by the layer/datatype number of name if no layer/datatype number is present.

The shapes moved will be added to any shapes already in the cells.

This method has been added in version 0.23.

(2) Signature: void **move_tree_shapes** (Cell source_cell, const CellMapping cell_mapping, const LayerMapping layer_mapping)

Description: Moves the shapes from the given cell and the cell tree below into this cell or subcells of this cell with layer mapping

source_cell:	The cell from where to move shapes and instances
cell_mapping:	The cell mapping object that determines how cells are identified between source and target layout

This method is provided if source and target cell reside in different layouts. If will move the shapes from all cells below the given source cell, but use a cell mapping object that provides a specification how cells are identified between the layouts. Cells in the source tree, for which no mapping is provided, will be flattened - their shapes will be propagated into parent cells for which a mapping is provided.

The cell mapping object provides various methods to map cell trees between layouts. See the CellMapping class for details about the mapping methods available. The cell mapping object is also responsible for creating a proper hierarchy of cells in the target layout if that is required.

In addition, the layer mapping object can be specified which maps source to target layers. This feature can be used to restrict the move operation to a subset of layers or to convert shapes to different layers in that step.

The shapes moved will be added to any shapes already in the cells.

This method has been added in version 0.23.

name

Signature: *[const]* string **name**

Description: Gets the cell's name

This may be an internal name for proxy cells. See basic_name for the formal name (PCell name or library cell name).

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string name)

Description: Renames the cell

Renaming a cell may cause name clashes, i.e. the name may be identical to the name of another cell. This does not have any immediate effect, but the cell needs to be renamed, for example when writing the layout to a GDS file.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'name'. This is the setter.

new

Signature: *[static]* new Cell ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

parent_cells

Signature: *[const]* unsigned long **parent_cells**

Description: Gets the number of parent cells

The number of parent cells (cells which reference our cell) is reported.

pcell_declaration

(1) Signature: *[const]* const PCellDeclaration ptr **pcell_declaration**

Description: Returns a reference to the PCell declaration

If this cell is not a PCell variant, this method returns nil. PCell variants are proxy cells which are PCell incarnations for a specific parameter set. The PCellDeclaration object allows one to retrieve PCell parameter definitions for example.

This method has been introduced in version 0.22.

(2) Signature: *[const]* const PCellDeclaration ptr **pcell_declaration** (const Instance instance)

Description: Returns the PCell declaration of a pcell instance

If the instance is not a PCell instance, this method returns nil. The PCellDeclaration object allows one to retrieve PCell parameter definitions for example.

This method has been introduced in version 0.22.

pcell_id

Signature: *[const]* unsigned int **pcell_id**

Description: Returns the PCell ID if the cell is a pcell variant

This method returns the ID which uniquely identifies the PCell within the layout where it's declared. It can be used to retrieve the PCell declaration or to create new PCell variants.

The method will be rarely used. It's more convenient to use pcell_declaration to directly retrieve the PCellDeclaration object for example.

This method has been introduced in version 0.22.

pcell_library

Signature: *[const]* Library ptr **pcell_library**

Description: Returns the library where the PCell is declared if this cell is a PCell and it is not defined locally.

A PCell often is not declared within the current layout but in some library. This method returns a reference to that library, which technically is the last of the chained library proxies. If this cell is not a PCell or it is not located in a library, this method returns nil.

This method has been introduced in version 0.22.

pcell_parameter

(1) Signature: *[const]* variant **pcell_parameter** (string name)

Description: Gets a PCell parameter by name if the cell is a PCell variant

If the cell is a PCell variant, this method returns the parameter with the given name. If the cell is not a PCell variant or the name is not a valid PCell parameter name, the return value is nil.

This method has been introduced in version 0.25.

(2) Signature: *[const]* variant **pcell_parameter** (const Instance instance, string name)

Description: Returns a PCell parameter by name for a pcell instance

If the given instance is a PCell instance, this method returns the value of the PCell parameter with the given name. If the instance is not a PCell instance or the name is not a valid PCell parameter name, this method returns nil.

This method has been introduced in version 0.25.

pcell_parameters

(1) Signature: *[const]* variant[] **pcell_parameters**

Description: Returns the PCell parameters for a pcell variant

If the cell is a PCell variant, this method returns a list of values for the PCell parameters. If the cell is not a PCell variant, this method returns an empty list. This method also returns the PCell parameters if the cell is a PCell imported from a library.

This method has been introduced in version 0.22.

(2) Signature: *[const]* variant[] **pcell_parameters** (const Instance instance)

Description: Returns the PCell parameters for a pcell instance

If the given instance is a PCell instance, this method returns a list of values for the PCell parameters. If the instance is not a PCell instance, this method returns an empty list.

This method has been introduced in version 0.22.

pcell_parameters_by_name

(1) Signature: *[const]* map<string,variant> **pcell_parameters_by_name**

Description: Returns the PCell parameters for a pcell variant as a name to value dictionary

If the cell is a PCell variant, this method returns a dictionary of values for the PCell parameters with the parameter names as the keys. If the cell is not a PCell variant, this method returns an empty dictionary. This method also returns the PCell parameters if the cell is a PCell imported from a library.

This method has been introduced in version 0.24.

(2) Signature: *[const]* map<string,variant> **pcell_parameters_by_name** (const Instance instance)

Description: Returns the PCell parameters for a pcell instance as a name to value dictionary

If the given instance is a PCell instance, this method returns a dictionary of values for the PCell parameters with the parameter names as the keys. If the instance is not a PCell instance, this method returns an empty dictionary.

This method has been introduced in version 0.24.

prop_id

Signature: *[const]* unsigned long **prop_id**

Description: Gets the properties ID associated with the cell

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'prop_id'. This is the getter.

prop_id=

Signature: void **prop_id=** (unsigned long id)

Description: Sets the properties ID associated with the cell

This method is provided, if a properties ID has been derived already. Usually it's more convenient to use delete_property, set_property or property.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'prop_id'. This is the setter.

property

Signature: variant **property** (variant key)

Description: Gets the user property with the given key

This method is a convenience method that gets the property with the given key. If no property with that key exists, it will return nil. Using that method is more convenient than using the layout object and the properties ID to retrieve the property value. This method has been introduced in version 0.23.

prune_cell

(1) Signature: void **prune_cell**

Description: Deletes the cell plus subcells not used otherwise

This deletes the cell and also all sub cells of the cell which are not used otherwise. All instances of this cell are deleted as well. A version of this method exists which allows one to specify the number of hierarchy levels to which subcells are considered.

After the cell has been deleted, the Cell object becomes invalid. Do not access methods or attributes of this object after deleting the cell.

This method has been introduced in version 0.23.

(2) Signature: void **prune_cell** (int levels)

Description: Deletes the cell plus subcells not used otherwise

levels:	The number of hierarchy levels to consider (-1: all, 0: none, 1: one level etc.)

This deletes the cell and also all sub cells of the cell which are not used otherwise. The number of hierarchy levels to consider can be specified as well. One level of hierarchy means that only the direct children of the cell are deleted with the cell itself. All instances of this cell are deleted as well.

After the cell has been deleted, the Cell object becomes invalid. Do not access methods or attributes of this object after deleting the cell.

This method has been introduced in version 0.23.

prune_subcells

(1) Signature: void **prune_subcells**

Description: Deletes all sub cells of the cell which are not used otherwise

This deletes all sub cells of the cell which are not used otherwise. All instances of the deleted cells are deleted as well. A version of this method exists which allows one to specify the number of hierarchy levels to which subcells are considered.

This method has been introduced in version 0.23.

(2) Signature: void **prune_subcells** (int levels)

Description: Deletes all sub cells of the cell which are not used otherwise down to the specified level of hierarchy

levels:	The number of hierarchy levels to consider (-1: all, 0: none, 1: one level etc.)

This deletes all sub cells of the cell which are not used otherwise. All instances of the deleted cells are deleted as well. It is possible to specify how many levels of hierarchy below the given root cell are considered.

This method has been introduced in version 0.23.

qname

Signature: *[const]* string **qname**

Description: Returns the library-qualified name

Library cells will be indicated by returning a qualified name composed of the library name, a dot and the basic cell name. For example: "Basic.TEXT" will be the qname of the TEXT cell of the Basic library. For non-library cells, the qname is identical to the basic name (see name).

This method has been introduced in version 0.25.

read

(1) Signature: unsigned int[] **read** (string file_name, const LoadLayoutOptions options)

Description: Reads a layout file into this cell

file_name:	The path of the file to read
options:	The reader options to use

Returns: The indexes of the cells created during the reading (new child cells)

The format of the file will be determined from the file name. The layout will be read into the cell, potentially creating new layers and a subhierarchy of cells below this cell.

This feature is equivalent to the following code:

```
def Cell.read(file_name, options)
  layout = RBA::Layout::new
  layout.read(file_name, options)
  cm = RBA::CellMapping::new
  cm.for_single_cell_full(self, layout.top_cell)
  self.move_tree_shapes(layout.top_cell)
end
```

See move_tree_shapes and CellMapping for more details and how to implement more elaborate schemes.

This method has been introduced in version 0.28.

(2) Signature: unsigned int[] **read** (string file_name)

Description: Reads a layout file into this cell

This version uses the default options for reading the file.

This method has been introduced in version 0.28.

refresh

Signature: void **refresh**

Description: Refreshes a proxy cell

If the cell is a PCell variant, this method recomputes the PCell. If the cell is a library proxy, this method reloads the information from the library, but not the library itself. Note that if the cell is an PCell variant for a PCell coming from a library, this method will not recompute the PCell. Instead, you can use Library#refresh to recompute all PCells from that library.

You can use Layout#refresh to refresh all cells from a layout.

This method has been introduced in version 0.22.

remove_meta_info

Signature: void **remove_meta_info** (string name)

Description: Removes meta information from the cell

See LayoutMetaInfo for details about cells and meta information.

This method has been introduced in version 0.28.8.

replace

(1) Signature: Instance **replace** (const Instance instance, const CellInstArray cell_inst_array)

Description: Replaces a cell instance (array) with a different one

Returns: An Instance object representing the new instance

This method has been introduced in version 0.16. It can only be used in editable mode. The instance given by the instance object (first argument) is replaced by the given instance (second argument). The new object will not have any properties.

(2) Signature: Instance **replace** (const Instance instance, const CellInstArray cell_inst_array, unsigned long property_id)

Description: Replaces a cell instance (array) with a different one with properties

Returns: An Instance object representing the new instance

This method has been introduced in version 0.16. It can only be used in editable mode. The instance given by the instance object (first argument) is replaced by the given instance (second argument) with the given properties Id. The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. The new object will not have any properties.

(3) Signature: Instance **replace** (const Instance instance, const DCellInstArray cell_inst_array)

Description: Replaces a cell instance (array) with a different one, given in micrometer units

Returns: An Instance object representing the new instance

This method is identical to the corresponding replace variant with a CellInstArray argument. It however accepts a micrometer-unit DCellInstArray object which is translated to database units internally.

This variant has been introduced in version 0.25.

(4) Signature: Instance **replace** (const Instance instance, const DCellInstArray cell_inst_array, unsigned long property_id)

Description: Replaces a cell instance (array) with a different one and new properties, where the cell instance is given in micrometer units

Returns: An Instance object representing the new instance

This method is identical to the corresponding replace variant with a CellInstArray argument and a property ID. It however accepts a micrometer-unit DCellInstArray object which is translated to database units internally.

This variant has been introduced in version 0.25.

replace_prop_id

Signature: Instance **replace_prop_id** (const Instance instance, unsigned long property_id)

Description: Replaces (or install) the properties of a cell

Returns: An Instance object representing the new instance

This method has been introduced in version 0.16. It can only be used in editable mode. Changes the properties Id of the given instance or install a properties Id on that instance if it does not have one yet. The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id.

set_property

Signature: void **set_property** (variant key, variant value)

Description: Sets the user property with the given key to the given value

This method is a convenience method that sets the property with the given key to the given value. If no property with that key exists, it will create one. Using that method is more convenient than creating a new property set with a new ID and assigning that properties ID. This method may change the properties ID. Note: GDS only supports integer keys. OASIS supports numeric and string keys. This method has been introduced in version 0.23.

shapes

(1) Signature: Shapes **shapes** (unsigned int layer_index)

Description: Returns the shapes list of the given layer

index: The layer index of the shapes list to retrieve

Returns: A reference to the shapes list

This method gives access to the shapes list on a certain layer. If the layer does not exist yet, it is created.

(2) Signature: *[const]* const Shapes ptr **shapes** (unsigned int layer_index)

Description: Returns the shapes list of the given layer (const version)

index: The layer index of the shapes list to retrieve

Returns: A reference to the shapes list

This method gives access to the shapes list on a certain layer. This is the const version - only const (reading) methods can be called on the returned object.

This variant has been introduced in version 0.26.4.

swap

Signature: void **swap** (unsigned int layer_index1, unsigned int layer_index2)

Description: Swaps the layers given

This method swaps two layers inside this cell.

transform

(1) Signature: Instance **transform** (const Instance instance, const Trans trans)

Description: Transforms the instance with the given transformation

Returns: A reference (an Instance object) to the new instance

This method has been introduced in version 0.16. The original instance may be deleted and re-inserted by this method. Therefore, a new reference is returned. It is permitted in editable mode only.

(2) Signature: Instance **transform** (const Instance instance, const ICplxTrans trans)

Description: Transforms the instance with the given complex integer transformation

Returns: A reference (an Instance object) to the new instance

This method has been introduced in version 0.23. The original instance may be deleted and re-inserted by this method. Therefore, a new reference is returned. It is permitted in editable mode only.

(3) Signature: Instance **transform** (const Instance instance, const DTrans trans)

Description: Transforms the instance with the transformation given in micrometer units

Returns: A reference (an Instance object) to the new instance

This method is identical to the corresponding transform method with a Trans argument. For this variant however, the transformation is given in micrometer units and is translated to database units internally.

This variant has been introduced in version 0.25.

(4) Signature: Instance **transform** (const Instance instance, const DCplxTrans trans)

Description: Transforms the instance with the given complex floating-point transformation given in micrometer units

Returns: A reference (an Instance object) to the new instance

This method is identical to the corresponding transform method with a ICplxTrans argument. For this variant however, the transformation is given in micrometer units and is translated to database units internally.

This variant has been introduced in version 0.25.

(5) Signature: void **transform** (const Trans trans)

Description: Transforms the cell by the given integer transformation

This method transforms all instances and all shapes by the given transformation. There is a variant called transform_into which applies the transformation to instances in a way such that it can be applied recursively to the child cells.

This method has been introduced in version 0.26.7.

(6) Signature: void **transform** (const ICplxTrans trans)

Description: Transforms the cell by the given complex integer transformation

This method transforms all instances and all shapes by the given transformation. There is a variant called transform_into which applies the transformation to instances in a way such that it can be applied recursively to the child cells. The difference is important in the presence of magnifications: "transform" will leave magnified instances while "transform_into" will not do so but expect the magnification to be applied inside the called cells too.

This method has been introduced in version 0.26.7.

(7) Signature: void **transform** (const DTrans trans)

Description: Transforms the cell by the given, micrometer-unit transformation

This method transforms all instances and all shapes by the given transformation. There is a variant called transform_into which applies the transformation to instances in a way such that it can be applied recursively to the child cells.

This method has been introduced in version 0.26.7.

(8) Signature: void **transform** (const DCplxTrans trans)

Description: Transforms the cell by the given, micrometer-unit transformation

This method transforms all instances and all shapes by the given transformation. There is a variant called transform_into which applies the transformation to instances in a way such that it can be applied recursively to the child cells. The difference is important in the presence of magnifications: "transform" will leave magnified instances while "transform_into" will not do so but expect the magnification to be applied inside the called cells too.

This method has been introduced in version 0.26.7.

transform_into

(1) Signature: Instance **transform_into** (const Instance instance, const Trans trans)

Description: Transforms the instance into a new coordinate system with the given transformation

Returns: A reference (an Instance object) to the new instance

In contrast to the transform method, this method allows propagation of the transformation into child cells. More precisely: it applies just a part of the given transformation to the instance, such that when transforming the cell instantiated and its shapes with the same transformation, the result will reflect the desired transformation. Mathematically spoken, the transformation of the instance (A) is transformed with the given transformation T using "A' = T * A * Tinv" where Tinv is the inverse of T. In effect, the transformation T commutes with the new instance transformation A' and can be applied to child cells as well. This method is therefore useful to transform a hierarchy of cells.

This method has been introduced in version 0.23. The original instance may be deleted and re-inserted by this method. Therefore, a new reference is returned. It is permitted in editable mode only.

(2) Signature: Instance **transform_into** (const Instance instance, const ICplxTrans trans)

Description: Transforms the instance into a new coordinate system with the given complex integer transformation

Returns: A reference (an Instance object) to the new instance

See the comments for the simple-transformation version for a description of this method. This method has been introduced in version 0.23. The original instance may be deleted and re-inserted by this method. Therefore, a new reference is returned. It is permitted in editable mode only.

(3) Signature: void **transform_into** (const Trans trans)

Description: Transforms the cell into a new coordinate system with the given transformation

This method transforms all instances and all shapes. The instances are transformed in a way that allows propagation of the transformation into child cells. For this, it applies just a part of the given transformation to the instance such that when transforming the shapes of the cell instantiated, the result will reflect the desired transformation. Mathematically spoken, the transformation of the instance (A) is transformed with the given transformation T using "A' = T * A * Tinv" where Tinv is the inverse of T. In effect, the transformation T commutes with the new instance transformation A' and can be applied to child cells as well. This method is therefore useful to transform a hierarchy of cells.

It has been introduced in version 0.23.

(4) Signature: void **transform_into** (const ICplxTrans trans)

Description: Transforms the cell into a new coordinate system with the given complex integer transformation

See the comments for the simple-transformation version for a description of this method. This method has been introduced in version 0.23.

(5) Signature: Instance **transform_into** (const Instance instance, const DTrans trans)

Description: Transforms the instance into a new coordinate system with the given transformation where the transformation is in micrometer units

Returns: A reference (an Instance object) to the new instance

This method is identical to the corresponding transform_into method with a Trans argument. For this variant however, the transformation is given in micrometer units and is translated to database units internally.

This variant has been introduced in version 0.25.

(6) Signature: Instance **transform_into** (const Instance instance, const DCplxTrans trans)

Description: Transforms the instance into a new coordinate system with the given complex transformation where the transformation is in micrometer units

Returns: A reference (an Instance object) to the new instance

This method is identical to the corresponding transform_into method with a ICplxTrans argument. For this variant however, the transformation is given in micrometer units and is translated to database units internally.

This variant has been introduced in version 0.25.

(7) Signature: void **transform_into** (const DTrans trans)

Description: Transforms the cell into a new coordinate system with the given transformation where the transformation is in micrometer units

This method is identical to the corresponding transform_into method with a Trans argument. For this variant however, the transformation is given in micrometer units and is translated to database units internally.

This variant has been introduced in version 0.25.

(8) Signature: void **transform_into** (const DCplxTrans trans)

Description: Transforms the cell into a new coordinate system with the given complex integer transformation where the transformation is in micrometer units

This method is identical to the corresponding transform_into method with a ICplxTrans argument. For this variant however, the transformation is given in micrometer units and is translated to database units internally.

This variant has been introduced in version 0.25.

write

(1) Signature: *[const]* void **write** (string file_name)

Description: Writes the cell to a layout file

The format of the file will be determined from the file name. Only the cell and its subtree below will be saved.

This method has been introduced in version 0.23.

(2) Signature: *[const]* void **write** (string file_name, const SaveLayoutOptions options)

Description: Writes the cell to a layout file

The format of the file will be determined from the file name. Only the cell and its subtree below will be saved. In contrast to the other 'write' method, this version allows one to specify save options, i.e. scaling etc.

This method has been introduced in version 0.23.

4.20. API reference - Class Instance

Notation used in Ruby API documentation

Module: db

Description: An instance proxy

An instance proxy is basically a pointer to an instance of different kinds, similar to Shape, the shape proxy. Instance objects can be duplicated without creating copies of the instances itself: the copy will still point to the same instance than the original.

When the Instance object is modified, the actual instance behind it is modified. The Instance object acts as a simplified interface for single and array instances with or without properties.

See The Database API for more details about the database objects.

Public constructors

new Instance ptr	**new**		Creates a new object of this class

Public methods

[const]	bool	**!=**	(const Instance b)	Tests for inequality of two Instance objects
[const]	bool	**<**	(const Instance b)	Provides an order criterion for two Instance objects
[const]	bool	**==**	(const Instance b)	Tests for equality of two Instance objects
[const]	variant	**[]**	(variant key)	Gets the user property with the given key or, if available, the PCell parameter with the name given by the key
	void	**[]=**	(variant key, variant value)	Sets the user property with the given key or, if available, the PCell parameter with the name given by the key
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	Vector	**a**		Returns the displacement vector for the 'a' axis
	void	**a=**	(const Vector a)	Sets the displacement vector for the 'a' axis

	void	**a=**	(const DVector a)	Sets the displacement vector for the 'a' axis in micrometer units
	void	**assign**	(const Instance other)	Assigns another object to self
[const]	Vector	**b**		Returns the displacement vector for the 'b' axis
	void	**b=**	(const Vector b)	Sets the displacement vector for the 'b' axis
	void	**b=**	(const DVector b)	Sets the displacement vector for the 'b' axis in micrometer units
[const]	Box	**bbox**		Gets the bounding box of the instance
[const]	Box	**bbox**	(unsigned int layer_index)	Gets the bounding box of the instance for a given layer
	Cell ptr	**cell**		Gets the Cell object of the cell this instance refers to
[const]	const Cell ptr	**cell**		Gets the Cell object of the cell this instance refers to
	void	**cell=**	(const Cell ptr cell)	Sets the Cell object this instance refers to
[const]	unsigned int	**cell_index**		Get the index of the cell this instance refers to
	void	**cell_index=**	(unsigned int cell_index)	Sets the index of the cell this instance refers to
[const]	CellInstArray	**cell_inst**		Gets the basic CellInstArray object associated with this instance reference.
	void	**cell_inst=**	(const CellInstArray inst)	Changes the CellInstArray object to the given one.
	void	**cell_inst=**	(const DCellInstArray inst)	Returns the basic cell instance array object by giving a micrometer unit object.
	void	**change_pcell_param**	(string name, variant value)	Changes a single parameter of a PCell instance to the given value
	void	**change_pcell_parameters**	(variant[] params)	Changes the parameters of a PCell instance to the list of parameters
	void	**change_pcell_param**	(map<string,variant> dict)	Changes the parameters of a PCell instance to the dictionary of parameters
	void	**convert_to_static**		Converts a PCell instance to a static cell
[const]	ICplxTrans	**cplx_trans**		Gets the complex transformation of the instance or the first instance in the array
	void	**cplx_trans=**	(const ICplxTrans t)	Sets the complex transformation of the instance or the first instance in the array

	void	**cplx_trans=**	(const DCplxTrans t)	Sets the complex transformation of the instance or the first instance in the array (in micrometer units)
[const]	DVector	**da**		Returns the displacement vector for the 'a' axis in micrometer units
	void	**da=**	(const DVector a)	Sets the displacement vector for the 'a' axis in micrometer units
[const]	DVector	**db**		Returns the displacement vector for the 'b' axis in micrometer units
	void	**db=**	(const DVector b)	Sets the displacement vector for the 'b' axis in micrometer units
[const]	DBox	**dbbox**		Gets the bounding box of the instance in micron units
[const]	DBox	**dbbox**	(unsigned int layer_index)	Gets the bounding box of the instance in micron units
	DCellInstArray	**dcell_inst**		Returns the micrometer unit version of the basic cell instance array object.
	void	**dcell_inst=**	(const DCellInstArray inst)	Returns the basic cell instance array object by giving a micrometer unit object.
[const]	DCplxTrans	**dcplx_trans**		Gets the complex transformation of the instance or the first instance in the array (in micrometer units)
	void	**dcplx_trans=**	(const DCplxTrans t)	Sets the complex transformation of the instance or the first instance in the array (in micrometer units)
	void	**delete**		Deletes this instance
	void	**delete_property**	(variant key)	Deletes the user property with the given key
[const]	DTrans	**dtrans**		Gets the transformation of the instance or the first instance in the array (in micrometer units)
	void	**dtrans=**	(const DTrans t)	Sets the transformation of the instance or the first instance in the array (in micrometer units)
[const]	new Instance ptr	**dup**		Creates a copy of self
	void	**explode**		Explodes the instance array
	void	**flatten**		Flattens the instance
	void	**flatten**	(int levels)	Flattens the instance
[const]	bool	**has_prop_id?**		Returns true, if the instance has properties
[const]	bool	**is_complex?**		Tests, if the array is a complex array
[const]	bool	**is_null?**		Checks, if the instance is a valid one

[const]	bool	**is_pcell?**		Returns a value indicating whether the instance is a PCell instance
[const]	bool	**is_regular_array?**		Tests, if this instance is a regular array
[const]	bool	**is_valid?**		Tests if the Instance object is still pointing to a valid instance
	Layout ptr	**layout**		Gets the layout this instance is contained in
[const]	const Layout ptr	**layout**		Gets the layout this instance is contained in
[const]	unsigned long	**na**		Returns the number of instances in the 'a' axis
	void	**na=**	(unsigned long na)	Sets the number of instances in the 'a' axis
[const]	unsigned long	**nb**		Returns the number of instances in the 'b' axis
	void	**nb=**	(unsigned long nb)	Sets the number of instances in the 'b' axis
	Cell ptr	**parent_cell**		Gets the cell this instance is contained in
[const]	const Cell ptr	**parent_cell**		Gets the cell this instance is contained in
	void	**parent_cell=**	(Cell ptr new_parent)	Moves the instance to a different cell
[const]	const PCellDeclaration ptr	**pcell_declaration**		Returns the PCell declaration object
[const]	variant	**pcell_parameter**	(string name)	Gets a PCell parameter by the name of the parameter
[const]	variant[]	**pcell_parameters**		Gets the parameters of a PCell instance as a list of values
[const]	map<string,variant>	**pcell_parameters_by_name**		Gets the parameters of a PCell instance as a dictionary of values vs. names
[const]	unsigned long	**prop_id**		Gets the properties ID associated with the instance
	void	**prop_id=**	(unsigned long id)	Sets the properties ID associated with the instance
[const]	variant	**property**	(variant key)	Gets the user property with the given key
	void	**set_property**	(variant key, variant value)	Sets the user property with the given key to the given value
[const]	unsigned long	**size**		Gets the number of single instances in the instance array
[const]	string	**to_s**		Creates a string showing the contents of the reference

[const]	string	**to_s**	(bool with_cellname)	Creates a string showing the contents of the reference
[const]	Trans	**trans**		Gets the transformation of the instance or the first instance in the array
	void	**trans=**	(const Trans t)	Sets the transformation of the instance or the first instance in the array
	void	**trans=**	(const DTrans t)	Sets the transformation of the instance or the first instance in the array (in micrometer units)
	void	**transform**	(const Trans t)	Transforms the instance array with the given transformation
	void	**transform**	(const ICplxTrans t)	Transforms the instance array with the given complex transformation
	void	**transform**	(const DTrans t)	Transforms the instance array with the given transformation (given in micrometer units)
	void	**transform**	(const DCplxTrans t)	Transforms the instance array with the given complex transformation (given in micrometer units)
	void	**transform_into**	(const Trans t)	Transforms the instance array with the given transformation
	void	**transform_into**	(const ICplxTrans t)	Transforms the instance array with the given transformation
	void	**transform_into**	(const DTrans t)	Transforms the instance array with the given transformation (given in micrometer units)
	void	**transform_into**	(const DCplxTrans t)	Transforms the instance array with the given complex transformation (given in micrometer units)

Deprecated methods (protected, public, static, non-static and constructors)

[const]	Box	**bbox_per_layer**	(unsigned int layer_index)	Use of this method is deprecated. Use bbox instead
	void	**create**		Use of this method is deprecated. Use _create instead
[const]	DBox	**dbbox_per_layer**	(unsigned int layer_index)	Use of this method is deprecated. Use dbbox instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=	**Signature**: *[const]* bool **!=** (const Instance b) **Description**: Tests for inequality of two Instance objects Warning: this operator returns true if both objects refer to the same instance, not just identical ones.
<	**Signature**: *[const]* bool **<** (const Instance b) **Description**: Provides an order criterion for two Instance objects Warning: this operator is just provided to establish any order, not a particular one.
==	**Signature**: *[const]* bool **==** (const Instance b) **Description**: Tests for equality of two Instance objects See the hint on the < operator.
[]	**Signature**: *[const]* variant **[]** (variant key) **Description**: Gets the user property with the given key or, if available, the PCell parameter with the name given by the key Getting the PCell parameter has priority over the user property. This method has been introduced in version 0.25.
[]=	**Signature**: void **[]=** (variant key, variant value) **Description**: Sets the user property with the given key or, if available, the PCell parameter with the name given by the key Setting the PCell parameter has priority over the user property. This method has been introduced in version 0.25.
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

a

Signature: *[const]* Vector **a**

Description: Returns the displacement vector for the 'a' axis

Starting with version 0.25 the displacement is of vector type.

Python specific notes:
The object exposes a readable attribute 'a'. This is the getter.

a=

(1) Signature: void **a=** (const Vector a)

Description: Sets the displacement vector for the 'a' axis

If the instance was not an array instance before it is made one.

This method has been introduced in version 0.23. Starting with version 0.25 the displacement is of vector type.

Python specific notes:
The object exposes a writable attribute 'a'. This is the setter.

(2) Signature: void **a=** (const DVector a)

Description: Sets the displacement vector for the 'a' axis in micrometer units

Like a= with an integer displacement, this method will set the displacement vector but it accepts a vector in micrometer units that is of DVector type. The vector will be translated to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'a'. This is the setter.
The object exposes a writable attribute 'da'. This is the setter.

assign

Signature: void **assign** (const Instance other)

Description: Assigns another object to self

b

Signature: *[const]* Vector **b**

Description: Returns the displacement vector for the 'b' axis

Starting with version 0.25 the displacement is of vector type.

Python specific notes:
The object exposes a readable attribute 'b'. This is the getter.

b=

(1) Signature: void **b=** (const Vector b)

Description: Sets the displacement vector for the 'b' axis

If the instance was not an array instance before it is made one.

This method has been introduced in version 0.23. Starting with version 0.25 the displacement is of vector type.

Python specific notes:
The object exposes a writable attribute 'b'. This is the setter.

(2) Signature: void **b=** (const DVector b)

Description: Sets the displacement vector for the 'b' axis in micrometer units

Like b= with an integer displacement, this method will set the displacement vector but it accepts a vector in micrometer units that is of DVector type. The vector will be translated to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'b'. This is the setter.
The object exposes a writable attribute 'db'. This is the setter.

bbox

(1) Signature: *[const]* Box **bbox**

Description: Gets the bounding box of the instance

The bounding box incorporates all instances that the array represents. It gives the overall extension of the child cell as seen in the calling cell (or all array members if the instance forms an array). This method has been introduced in version 0.23.

(2) Signature: *[const]* Box **bbox** (unsigned int layer_index)

Description: Gets the bounding box of the instance for a given layer

layer_index:	The index of the layer the bounding box will be computed for.

The bounding box incorporates all instances that the array represents. It gives the overall extension of the child cell as seen in the calling cell (or all array members if the instance forms an array) for the given layer. If the layer is empty in this cell and all its children', an empty bounding box will be returned. This method has been introduced in version 0.25. 'bbox' is the preferred synonym for it since version 0.28.

bbox_per_layer

Signature: *[const]* Box **bbox_per_layer** (unsigned int layer_index)

Description: Gets the bounding box of the instance for a given layer

layer_index:	The index of the layer the bounding box will be computed for.

Use of this method is deprecated. Use bbox instead

The bounding box incorporates all instances that the array represents. It gives the overall extension of the child cell as seen in the calling cell (or all array members if the instance forms an array) for the given layer. If the layer is empty in this cell and all its children', an empty bounding box will be returned. This method has been introduced in version 0.25. 'bbox' is the preferred synonym for it since version 0.28.

cell

(1) Signature: Cell ptr **cell**

Description: Gets the Cell object of the cell this instance refers to

Please note that before version 0.23 this method returned the cell the instance is contained in. For consistency, this method has been renamed parent_cell.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'cell'. This is the getter.

(2) Signature: *[const]* const Cell ptr **cell**

Description: Gets the Cell object of the cell this instance refers to

This is the const version of the cell method. It will return a const Cell object and itself can be called on a const Instance object.

This variant has been introduced in version 0.25.

Python specific notes:
This method is available as 'cell_' in Python to distiguish it from the property with the same name.

cell=

Signature: void **cell=** (const Cell ptr cell)

Description: Sets the Cell object this instance refers to

Setting the cell object to nil is equivalent to deleting the instance.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'cell'. This is the setter.

cell_index

Signature: *[const]* unsigned int **cell_index**

Description: Get the index of the cell this instance refers to

Python specific notes:
The object exposes a readable attribute 'cell_index'. This is the getter.

cell_index=

Signature: void **cell_index=** (unsigned int cell_index)

Description: Sets the index of the cell this instance refers to

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'cell_index'. This is the setter.

cell_inst

Signature: *[const]* CellInstArray **cell_inst**

Description: Gets the basic CellInstArray object associated with this instance reference.

Python specific notes:
The object exposes a readable attribute 'cell_inst'. This is the getter.

cell_inst=

(1) Signature: void **cell_inst=** (const CellInstArray inst)

Description: Changes the CellInstArray object to the given one.

This method replaces the instance by the given CellInstArray object.

This method has been introduced in version 0.22

Python specific notes:
The object exposes a writable attribute 'cell_inst'. This is the setter.

(2) Signature: void **cell_inst=** (const DCellInstArray inst)

Description: Returns the basic cell instance array object by giving a micrometer unit object.

This method replaces the instance by the given CellInstArray object and it internally transformed into database units.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a writable attribute 'cell_inst'. This is the setter.
The object exposes a writable attribute 'dcell_inst'. This is the setter.

change_pcell_parameter

Signature: void **change_pcell_parameter** (string name, variant value)

Description: Changes a single parameter of a PCell instance to the given value

This method changes a parameter of a PCell instance to the given value. The name identifies the PCell parameter and must correspond to one parameter listed in the PCell declaration.

This method has been introduced in version 0.24.

change_pcell_parameters

(1) Signature: void **change_pcell_parameters** (variant[] params)

Description: Changes the parameters of a PCell instance to the list of parameters

This method changes the parameters of a PCell instance to the given list of parameters. The list must correspond to the parameters listed in the pcell declaration. A more convenient method is provided with the same name which accepts a dictionary of names and values . This method has been introduced in version 0.24.

(2) Signature: void **change_pcell_parameters** (map<string,variant> dict)

Description: Changes the parameters of a PCell instance to the dictionary of parameters

This method changes the parameters of a PCell instance to the given values. The values are specifies as a dictionary of names (keys) vs. values. Unknown names are ignored and only the parameters listed in the dictionary are changed.

This method has been introduced in version 0.24.

convert_to_static

Signature: void **convert_to_static**

Description: Converts a PCell instance to a static cell

If the instance is a PCell instance, this method will convert the cell into a static cell and remove the PCell variant if required. A new cell will be created containing the PCell content but being a static cell. If the instance is not a PCell instance, this method won't do anything.

This method has been introduced in version 0.24.

cplx_trans

Signature: *[const]* ICplxTrans **cplx_trans**

Description: Gets the complex transformation of the instance or the first instance in the array

This method is always valid compared to trans, since simple transformations can be expressed as complex transformations as well.

Python specific notes:
The object exposes a readable attribute 'cplx_trans'. This is the getter.

cplx_trans=

(1) Signature: void **cplx_trans=** (const ICplxTrans t)

Description: Sets the complex transformation of the instance or the first instance in the array

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'cplx_trans'. This is the setter.

(2) Signature: void **cplx_trans=** (const DCplxTrans t)

Description: Sets the complex transformation of the instance or the first instance in the array (in micrometer units)

This method sets the transformation the same way as cplx_trans=, but the displacement of this transformation is given in micrometer units. It is internally translated into database units.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'cplx_trans'. This is the setter.
The object exposes a writable attribute 'dcplx_trans'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

da

Signature: *[const]* DVector **da**

Description: Returns the displacement vector for the 'a' axis in micrometer units

Like a, this method returns the displacement, but it will be translated to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'da'. This is the getter.

da=

Signature: void **da=** (const DVector a)

Description: Sets the displacement vector for the 'a' axis in micrometer units

Like a= with an integer displacement, this method will set the displacement vector but it accepts a vector in micrometer units that is of DVector type. The vector will be translated to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'a'. This is the setter.
The object exposes a writable attribute 'da'. This is the setter.

db

Signature: *[const]* DVector **db**

Description: Returns the displacement vector for the 'b' axis in micrometer units

Like b, this method returns the displacement, but it will be translated to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'db'. This is the getter.

db=

Signature: void **db=** (const DVector b)

Description: Sets the displacement vector for the 'b' axis in micrometer units

Like b= with an integer displacement, this method will set the displacement vector but it accepts a vector in micrometer units that is of DVector type. The vector will be translated to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'b'. This is the setter.
The object exposes a writable attribute 'db'. This is the setter.

dbbox

(1) Signature: *[const]* DBox **dbbox**

Description: Gets the bounding box of the instance in micron units

Gets the bounding box (see bbox) of the instance, but will compute the micrometer unit box by multiplying bbox with the database unit.

This method has been introduced in version 0.25.

(2) Signature: *[const]* DBox **dbbox** (unsigned int layer_index)

Description: Gets the bounding box of the instance in micron units

layer_index:	The index of the layer the bounding box will be computed for.

Gets the bounding box (see bbox) of the instance, but will compute the micrometer unit box by multiplying bbox with the database unit.

This method has been introduced in version 0.25. 'dbbox' is the preferred synonym for it since version 0.28.

dbbox_per_layer

Signature: *[const]* DBox **dbbox_per_layer** (unsigned int layer_index)

Description: Gets the bounding box of the instance in micron units

layer_index:	The index of the layer the bounding box will be computed for.

Use of this method is deprecated. Use dbbox instead

Gets the bounding box (see bbox) of the instance, but will compute the micrometer unit box by multiplying bbox with the database unit.

This method has been introduced in version 0.25. 'dbbox' is the preferred synonym for it since version 0.28.

dcell_inst

Signature: DCellInstArray **dcell_inst**

Description: Returns the micrometer unit version of the basic cell instance array object.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'dcell_inst'. This is the getter.

dcell_inst=

Signature: void **dcell_inst=** (const DCellInstArray inst)

Description: Returns the basic cell instance array object by giving a micrometer unit object.

This method replaces the instance by the given CellInstArray object and it internally transformed into database units.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a writable attribute 'cell_inst'. This is the setter.
The object exposes a writable attribute 'dcell_inst'. This is the setter.

dcplx_trans

Signature: *[const]* DCplxTrans **dcplx_trans**

Description: Gets the complex transformation of the instance or the first instance in the array (in micrometer units)

This method returns the same transformation as cplx_trans, but the displacement of this transformation is given in micrometer units. It is internally translated from database units into micrometers.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'dcplx_trans'. This is the getter.

dcplx_trans=

Signature: void **dcplx_trans=** (const DCplxTrans t)

Description: Sets the complex transformation of the instance or the first instance in the array (in micrometer units)

This method sets the transformation the same way as cplx_trans=, but the displacement of this transformation is given in micrometer units. It is internally translated into database units.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'cplx_trans'. This is the setter.
The object exposes a writable attribute 'dcplx_trans'. This is the setter.

delete

Signature: void **delete**

Description: Deletes this instance

After this method was called, the instance object is pointing to nothing.

This method has been introduced in version 0.23.

delete_property

Signature: void **delete_property** (variant key)

Description: Deletes the user property with the given key

This method is a convenience method that deletes the property with the given key. It does nothing if no property with that key exists. Using that method is more convenient than creating a new property set with a new ID and assigning that properties ID. This method may change the properties ID. Calling this method may invalidate any iterators. It should not be called inside a loop iterating over instances.

This method has been introduced in version 0.22.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dtrans

Signature: *[const]* DTrans **dtrans**

Description: Gets the transformation of the instance or the first instance in the array (in micrometer units)

This method returns the same transformation as cplx_trans, but the displacement of this transformation is given in micrometer units. It is internally translated from database units into micrometers.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'dtrans'. This is the getter.

dtrans=

Signature: void **dtrans=** (const DTrans t)

Description: Sets the transformation of the instance or the first instance in the array (in micrometer units)

This method sets the transformation the same way as cplx_trans=, but the displacement of this transformation is given in micrometer units. It is internally translated into database units.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.
The object exposes a writable attribute 'dtrans'. This is the setter.

dup

Signature: *[const]* new Instance ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

explode

Signature: void **explode**

Description: Explodes the instance array

This method does nothing if the instance was not an array before. The instance object will point to the first instance of the array afterwards.

This method has been introduced in version 0.23.

flatten

(1) Signature: void **flatten**

Description: Flattens the instance

This method will convert the instance to a number of shapes which are equivalent to the content of the cell. The instance itself will be removed. There is another variant of this method which allows specification of the number of hierarchy levels to flatten.

This method has been introduced in version 0.24.

(2) Signature: void **flatten** (int levels)

Description: Flattens the instance

This method will convert the instance to a number of shapes which are equivalent to the content of the cell. The instance itself will be removed. This version of the method allows specification of the number of hierarchy levels to remove. Specifying 1 for 'levels' will remove the instance and replace it by the contents of the cell. Specifying a negative value or zero for the number of levels will flatten the instance completely.

This method has been introduced in version 0.24.

has_prop_id?

Signature: *[const]* bool **has_prop_id?**

Description: Returns true, if the instance has properties

is_complex?

Signature: *[const]* bool **is_complex?**

Description: Tests, if the array is a complex array

Returns true if the array represents complex instances (that is, with magnification and arbitrary rotation angles).

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_null?

Signature: *[const]* bool **is_null?**

Description: Checks, if the instance is a valid one

is_pcell?

Signature: *[const]* bool **is_pcell?**

Description: Returns a value indicating whether the instance is a PCell instance

This method has been introduced in version 0.24.

is_regular_array?

Signature: *[const]* bool **is_regular_array?**

Description: Tests, if this instance is a regular array

is_valid?

Signature: *[const]* bool **is_valid?**

Description: Tests if the Instance object is still pointing to a valid instance

If the instance represented by the given reference has been deleted, this method returns false. If however, another instance has been inserted already that occupies the original instances position, this method will return true again.

This method has been introduced in version 0.23 and is a shortcut for "inst.cell.is_valid?(inst)".

layout

(1) Signature: Layout ptr **layout**

Description: Gets the layout this instance is contained in

This method has been introduced in version 0.22.

(2) Signature: *[const]* const Layout ptr **layout**

Description: Gets the layout this instance is contained in

This const version of the method has been introduced in version 0.25.

na

Signature: *[const]* unsigned long **na**

Description: Returns the number of instances in the 'a' axis

Python specific notes:
The object exposes a readable attribute 'na'. This is the getter.

na=

Signature: void **na=** (unsigned long na)

Description: Sets the number of instances in the 'a' axis

If the instance was not an array instance before it is made one.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'na'. This is the setter.

nb

Signature: *[const]* unsigned long **nb**

Description: Returns the number of instances in the 'b' axis

Python specific notes:
The object exposes a readable attribute 'nb'. This is the getter.

nb=

Signature: void **nb=** (unsigned long nb)

Description: Sets the number of instances in the 'b' axis

If the instance was not an array instance before it is made one.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'nb'. This is the setter.

new

Signature: *[static]* new Instance ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

parent_cell

(1) Signature: Cell ptr **parent_cell**

Description: Gets the cell this instance is contained in

Returns nil if the instance does not live inside a cell. This method was named "cell" previously which lead to confusion with cell_index. It was renamed to "parent_cell" in version 0.23.

Python specific notes:
The object exposes a readable attribute 'parent_cell'. This is the getter.

(2) Signature: *[const]* const Cell ptr **parent_cell**

Description: Gets the cell this instance is contained in

Returns nil if the instance does not live inside a cell.

This const version of the parent_cell method has been introduced in version 0.25.

Python specific notes:
This method is available as 'parent_cell_' in Python to distiguish it from the property with the same name.

parent_cell=

Signature: void **parent_cell=** (Cell ptr new_parent)

Description: Moves the instance to a different cell

Both the current and the target cell must live in the same layout.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'parent_cell'. This is the setter.

pcell_declaration

Signature: *[const]* const PCellDeclaration ptr **pcell_declaration**

Description: Returns the PCell declaration object

If the instance is a PCell instance, this method returns the PCell declaration object for that PCell. If not, this method will return nil. This method has been introduced in version 0.24.

pcell_parameter

Signature: *[const]* variant **pcell_parameter** (string name)

Description: Gets a PCell parameter by the name of the parameter

Returns: The parameter value or nil if the instance is not a PCell or does not have a parameter with given name

This method has been introduced in version 0.25.

pcell_parameters

Signature: *[const]* variant[] **pcell_parameters**

Description: Gets the parameters of a PCell instance as a list of values

Returns: A list of values

If the instance is a PCell instance, this method will return an array of values where each value corresponds to one parameter. The order of the values is the order the parameters are declared in the PCell declaration. If the instance is not a PCell instance, this list returned will be empty.

This method has been introduced in version 0.24.

pcell_parameters_by_name

Signature: *[const]* map<string,variant> **pcell_parameters_by_name**

Description: Gets the parameters of a PCell instance as a dictionary of values vs. names

Returns: A dictionary of values by parameter name

If the instance is a PCell instance, this method will return a map of values vs. parameter names. The names are the ones defined in the PCell declaration.If the instance is not a PCell instance, the dictionary returned will be empty.

This method has been introduced in version 0.24.

prop_id

Signature: *[const]* unsigned long **prop_id**

Description: Gets the properties ID associated with the instance

Python specific notes:
The object exposes a readable attribute 'prop_id'. This is the getter.

prop_id=

Signature: void **prop_id=** (unsigned long id)

Description: Sets the properties ID associated with the instance

This method is provided, if a properties ID has been derived already. Usually it's more convenient to use delete_property, set_property or property.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'prop_id'. This is the setter.

property

Signature: *[const]* variant **property** (variant key)

Description: Gets the user property with the given key

This method is a convenience method that gets the property with the given key. If no property with that key exists, it will return nil. Using that method is more convenient than using the layout object and the properties ID to retrieve the property value. This method has been introduced in version 0.22.

set_property

Signature: void **set_property** (variant key, variant value)

Description: Sets the user property with the given key to the given value

This method is a convenience method that sets the property with the given key to the given value. If no property with that key exists, it will create one. Using that method is more convenient than creating a new property set with a new ID and assigning that properties ID. This method may change the properties ID. Note: GDS only supports integer keys. OASIS supports numeric and string keys. Calling this method may invalidate any iterators. It should not be called inside a loop iterating over instances.

This method has been introduced in version 0.22.

size

Signature: *[const]* unsigned long **size**

Description: Gets the number of single instances in the instance array

If the instance represents a single instance, the count is 1. Otherwise it is na*nb.

Python specific notes:
This method is also available as 'len(object)'.

to_s

(1) Signature: *[const]* string **to_s**

Description: Creates a string showing the contents of the reference

This method has been introduced with version 0.16.

Python specific notes:
This method is also available as 'str(object)'.

(2) Signature: *[const]* string **to_s** (bool with_cellname)

Description: Creates a string showing the contents of the reference

Passing true to with_cellname makes the string contain the cellname instead of the cell index

This method has been introduced with version 0.23.

trans

Signature: *[const]* Trans **trans**

Description: Gets the transformation of the instance or the first instance in the array

The transformation returned is only valid if the array does not represent a complex transformation array

Python specific notes:
The object exposes a readable attribute 'trans'. This is the getter.

trans=

(1) Signature: void **trans=** (const Trans t)

Description: Sets the transformation of the instance or the first instance in the array

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.

(2) Signature: void **trans=** (const DTrans t)

Description: Sets the transformation of the instance or the first instance in the array (in micrometer units)

This method sets the transformation the same way as cplx_trans=, but the displacement of this transformation is given in micrometer units. It is internally translated into database units.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.
The object exposes a writable attribute 'dtrans'. This is the setter.

transform

(1) Signature: void **transform** (const Trans t)

Description: Transforms the instance array with the given transformation

See Cell#transform for a description of this method.

This method has been introduced in version 0.23.

(2) Signature: void **transform** (const ICplxTrans t)

Description: Transforms the instance array with the given complex transformation

See Cell#transform for a description of this method.

This method has been introduced in version 0.23.

(3) Signature: void **transform** (const DTrans t)

Description: Transforms the instance array with the given transformation (given in micrometer units)

Transforms the instance like transform does, but with a transformation given in micrometer units. The displacement of this transformation is given in micrometers and is internally translated to database units.

This method has been introduced in version 0.25.

(4) Signature: void **transform** (const DCplxTrans t)

Description: Transforms the instance array with the given complex transformation (given in micrometer units)

Transforms the instance like transform does, but with a transformation given in micrometer units. The displacement of this transformation is given in micrometers and is internally translated to database units.

This method has been introduced in version 0.25.

transform_into

(1) Signature: void **transform_into** (const Trans t)

Description: Transforms the instance array with the given transformation

See Cell#transform_into for a description of this method.

This method has been introduced in version 0.23.

(2) Signature: void **transform_into** (const ICplxTrans t)

Description: Transforms the instance array with the given transformation

See Cell#transform_into for a description of this method.

This method has been introduced in version 0.23.

(3) Signature: void **transform_into** (const DTrans t)

Description: Transforms the instance array with the given transformation (given in micrometer units)

Transforms the instance like transform_into does, but with a transformation given in micrometer units. The displacement of this transformation is given in micrometers and is internally translated to database units.

This method has been introduced in version 0.25.

(4) Signature: void **transform_into** (const DCplxTrans t)

Description: Transforms the instance array with the given complex transformation (given in micrometer units)

Transforms the instance like transform_into does, but with a transformation given in micrometer units. The displacement of this transformation is given in micrometers and is internally translated to database units.

This method has been introduced in version 0.25.

4.21. API reference - Class ParentInstArray

Notation used in Ruby API documentation

Module: db

Description: A parent instance

A parent instance is basically an inverse instance: instead of pointing to the child cell, it is pointing to the parent cell and the transformation is representing the shift of the parent cell relative to the child cell. For memory performance, a parent instance is not stored as a instance but rather as a reference to a child instance and a reference to the cell which is the parent. The parent instance itself is computed on the fly. It is representative for a set of instances belonging to the same cell index. The special parent instance iterator takes care of producing the right sequence (Cell#each_parent_inst).

See The Database API for more details about the database objects.

Public constructors

new ParentInstArray ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const ParentIns other)	Assigns another object to self
[const]	Instance	**child_inst**		Retrieve the child instance associated with this parent instance
[const]	DCellInstArray	**dinst**		Compute the inverse instance by which the parent is seen from the child in micrometer units
[const]	new ParentInstArray ptr	**dup**		Creates a copy of self
[const]	CellInstArray	**inst**		Compute the inverse instance by which the parent is seen from the child
[const]	unsigned int	**parent_cell_index**		Gets the index of the parent cell

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const ParentInstArray other)

Description: Assigns another object to self

child_inst

Signature: *[const]* Instance **child_inst**

Description: Retrieve the child instance associated with this parent instance

Starting with version 0.15, this method returns an Instance object rather than a CellInstArray reference.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dinst

Signature: *[const]* DCellInstArray **dinst**

Description: Compute the inverse instance by which the parent is seen from the child in micrometer units

This convenience method has been introduced in version 0.28.

dup

Signature: *[const]* new ParentInstArray ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

inst

Signature: *[const]* CellInstArray **inst**

Description: Compute the inverse instance by which the parent is seen from the child

is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**Signature**: *[static]* new ParentInstArray ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
parent_cell_index	**Signature**: *[const]* unsigned int **parent_cell_index** **Description**: Gets the index of the parent cell

4.22. API reference - Class CellInstArray

Notation used in Ruby API documentation

Module: db

Description: A single or array cell instance

This object represents either single or array cell instances. A cell instance array is a regular array, described by two displacement vectors (a, b) and the instance count along that axes (na, nb).

In addition, this object represents either instances with simple transformations or instances with complex transformations. The latter includes magnified instances and instances rotated by an arbitrary angle.

The cell which is instantiated is given by a cell index. The cell index can be converted to a cell pointer by using Layout#cell. The cell index of a cell can be obtained using Cell#cell_index.

See The Database API for more details about the database objects.

Public constructors

new CellInstArray ptr	**new**		Creates en empty cell instance with size 0
new CellInstArray ptr	**new**	(unsigned int cell_index, const Trans trans)	Creates a single cell instance
new CellInstArray ptr	**new**	(const Cell ptr cell, const Trans trans)	Creates a single cell instance
new CellInstArray ptr	**new**	(unsigned int cell_index, const Vector disp)	Creates a single cell instance
new CellInstArray ptr	**new**	(const Cell ptr cell, const Vector disp)	Creates a single cell instance
new CellInstArray ptr	**new**	(unsigned int cell_index, const ICplxTrans trans)	Creates a single cell instance with a complex transformation
new CellInstArray ptr	**new**	(const Cell ptr cell, const ICplxTrans trans)	Creates a single cell instance with a complex transformation
new CellInstArray ptr	**new**	(unsigned int cell_index, const Trans trans, const Vector a, const Vector b, unsigned long na, unsigned long nb)	Creates a single cell instance
new CellInstArray ptr	**new**	(const Cell ptr cell, const Trans trans, const Vector a, const Vector b, unsigned long na, unsigned long nb)	Creates a single cell instance
new CellInstArray ptr	**new**	(unsigned int cell_index, const Vector disp, const Vector a, const Vector b, unsigned long na, unsigned long nb)	Creates a single cell instance

new CellInstArray ptr	new	(const Cell ptr cell, const Vector disp, const Vector a, const Vector b, unsigned long na, unsigned long nb)	Creates a single cell instance
new CellInstArray ptr	new	(unsigned int cell_index, const ICplxTrans trans, const Vector a, const Vector b, unsigned long na, unsigned long nb)	Creates a single cell instance with a complex transformation
new CellInstArray ptr	new	(const Cell ptr cell, const ICplxTrans trans, const Vector a, const Vector b, unsigned long na, unsigned long nb)	Creates a single cell instance with a complex transformation

Public methods

[const]	bool	!=	(const CellInstArray other)	Compares two arrays for inequality
[const]	bool	<	(const CellInstArray other)	Compares two arrays for 'less'
[const]	bool	==	(const CellInstArray other)	Compares two arrays for equality
	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	Vector	a		Gets the displacement vector for the 'a' axis
	void	a=	(const Vector vector)	Sets the displacement vector for the 'a' axis
	void	assign	(const CellInstArray other)	Assigns another object to self

[const]	Vector	**b**		Gets the displacement vector for the 'b' axis
	void	**b=**	(const Vector vector)	Sets the displacement vector for the 'b' axis
[const]	Box	**bbox**	(const Layout layout, unsigned int layer_index)	Gets the bounding box of the array with respect to one layer
[const]	Box	**bbox**	(const Layout layout)	Gets the bounding box of the array
	void	**cell=**	(Cell ptr cell)	Sets the cell this instance refers to
[const]	unsigned int	**cell_index**		Gets the cell index of the cell instantiated
	void	**cell_index=**	(unsigned int index)	Sets the index of the cell this instance refers to
[const]	ICplxTrans	**cplx_trans**		Gets the complex transformation of the first instance in the array
	void	**cplx_trans=**	(const ICplxTrans trans)	Sets the complex transformation of the instance or the first instance in the array
[const]	new CellInstArray ptr	**dup**		Creates a copy of self
[const,iter]	ICplxTrans	**each_cplx_trans**		Gets the complex transformations represented by this instance
[const,iter]	Trans	**each_trans**		Gets the simple transformations represented by this instance
[const]	unsigned long	**hash**		Computes a hash value
	void	**invert**		Inverts the array reference
[const]	bool	**is_complex?**		Gets a value indicating whether the array is a complex array
[const]	bool	**is_regular_array?**		Gets a value indicating whether this instance is a regular array
[const]	unsigned long	**na**		Gets the number of instances in the 'a' axis
	void	**na=**	(unsigned long n)	Sets the number of instances in the 'a' axis
[const]	unsigned long	**nb**		Gets the number of instances in the 'b' axis
	void	**nb=**	(unsigned long n)	Sets the number of instances in the 'b' axis
[const]	unsigned long	**size**		Gets the number of single instances in the array
[const]	string	**to_s**		Converts the array to a string

[const]	Trans	**trans**		Gets the transformation of the first instance in the array
	void	**trans=**	(const Trans t)	Sets the transformation of the instance or the first instance in the array
	void	**transform**	(const Trans trans)	Transforms the cell instance with the given transformation
	void	**transform**	(const ICplxTrans trans)	Transforms the cell instance with the given complex transformation
[const]	CellInstArray	**transformed**	(const Trans trans)	Gets the transformed cell instance
[const]	CellInstArray	**transformed**	(const ICplxTrans trans)	Gets the transformed cell instance (complex transformation)

Deprecated methods (protected, public, static, non-static and constructors)

[const]	Box	**bbox_per_layer**	(const Layout layout, unsigned int layer_index)	Use of this method is deprecated. Use bbox instead
	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=	**Signature**: *[const]* bool **!=** (const CellInstArray other) **Description**: Compares two arrays for inequality
<	**Signature**: *[const]* bool **<** (const CellInstArray other) **Description**: Compares two arrays for 'less' The comparison provides an arbitrary sorting criterion and not specific sorting order. It is guaranteed that if an array a is less than b, b is not less than a. In addition, it a is not less than b and b is not less than a, then a is equal to b.
==	**Signature**: *[const]* bool **==** (const CellInstArray other) **Description**: Compares two arrays for equality

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

a

Signature: *[const]* Vector **a**

Description: Gets the displacement vector for the 'a' axis

Starting with version 0.25 the displacement is of vector type.

Python specific notes:
The object exposes a readable attribute 'a'. This is the getter.

a=

Signature: void **a=** (const Vector vector)

Description: Sets the displacement vector for the 'a' axis

If the instance was not regular before this property is set, it will be initialized to a regular instance.

This method was introduced in version 0.22. Starting with version 0.25 the displacement is of vector type.

Python specific notes:
The object exposes a writable attribute 'a'. This is the setter.

assign

Signature: void **assign** (const CellInstArray other)

Description: Assigns another object to self

b

Signature: *[const]* Vector **b**

Description: Gets the displacement vector for the 'b' axis

Starting with version 0.25 the displacement is of vector type.

Python specific notes:
The object exposes a readable attribute 'b'. This is the getter.

b=

Signature: void **b=** (const Vector vector)

Description: Sets the displacement vector for the 'b' axis

If the instance was not regular before this property is set, it will be initialized to a regular instance.

This method was introduced in version 0.22. Starting with version 0.25 the displacement is of vector type.

Python specific notes:
The object exposes a writable attribute 'b'. This is the setter.

bbox

(1) Signature: *[const]* Box **bbox** (const Layout layout, unsigned int layer_index)

Description: Gets the bounding box of the array with respect to one layer

The bounding box incorporates all instances that the array represents. It needs the layout object to access the actual cell from the cell index.

'bbox' is the preferred synonym since version 0.28.

(2) Signature: *[const]* Box **bbox** (const Layout layout)

Description: Gets the bounding box of the array

The bounding box incorporates all instances that the array represents. It needs the layout object to access the actual cell from the cell index.

bbox_per_layer

Signature: *[const]* Box **bbox_per_layer** (const Layout layout, unsigned int layer_index)

Description: Gets the bounding box of the array with respect to one layer

Use of this method is deprecated. Use bbox instead

The bounding box incorporates all instances that the array represents. It needs the layout object to access the actual cell from the cell index.

'bbox' is the preferred synonym since version 0.28.

cell=

Signature: void **cell=** (Cell ptr cell)

Description: Sets the cell this instance refers to

This is a convenience method and equivalent to 'cell_index = cell.cell_index()'. There is no getter for the cell pointer because the CellInstArray object only knows about cell indexes.

This convenience method has been introduced in version 0.28.

Python specific notes:
The object exposes a writable attribute 'cell'. This is the setter.

cell_index

Signature: *[const]* unsigned int **cell_index**

Description: Gets the cell index of the cell instantiated

Use Layout#cell to get the Cell object from the cell index.

Python specific notes:
The object exposes a readable attribute 'cell_index'. This is the getter.

cell_index=

Signature: void **cell_index=** (unsigned int index)

Description: Sets the index of the cell this instance refers to

Python specific notes:
The object exposes a writable attribute 'cell_index'. This is the setter.

cplx_trans

Signature: *[const]* ICplxTrans **cplx_trans**

Description: Gets the complex transformation of the first instance in the array

This method is always applicable, compared to trans, since simple transformations can be expressed as complex transformations as well.

Python specific notes:
The object exposes a readable attribute 'cplx_trans'. This is the getter.

cplx_trans=

Signature: void **cplx_trans=** (const ICplxTrans trans)

Description: Sets the complex transformation of the instance or the first instance in the array

This method was introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'cplx_trans'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new CellInstArray ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_cplx_trans

Signature: *[const,iter]* ICplxTrans **each_cplx_trans**

Description: Gets the complex transformations represented by this instance

For a single instance, this iterator will deliver the single, complex transformation. For array instances, the iterator will deliver each complex transformation of the expanded array. This iterator is a generalization of each_trans for general complex transformations.

This method has been introduced in version 0.25.

each_trans

Signature: *[const,iter]* Trans **each_trans**

Description: Gets the simple transformations represented by this instance

For a single instance, this iterator will deliver the single, simple transformation. For array instances, the iterator will deliver each simple transformation of the expanded array.

This iterator will only deliver valid transformations if the instance array is not of complex type (see is_complex?). A more general iterator that delivers the complex transformations is each_cplx_trans.

This method has been introduced in version 0.25.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given cell instance. This method enables cell instances as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

invert

Signature: void **invert**

Description: Inverts the array reference

The inverted array reference describes in which transformations the parent cell is seen from the current cell.

is_complex?

Signature: *[const]* bool **is_complex?**

Description: Gets a value indicating whether the array is a complex array

Returns true if the array represents complex instances (that is, with magnification and arbitrary rotation angles).

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_regular_array?

Signature: *[const]* bool **is_regular_array?**

Description: Gets a value indicating whether this instance is a regular array

na

Signature: *[const]* unsigned long **na**

Description: Gets the number of instances in the 'a' axis

Python specific notes:
The object exposes a readable attribute 'na'. This is the getter.

na=

Signature: void **na=** (unsigned long n)

Description: Sets the number of instances in the 'a' axis

If the instance was not regular before this property is set to a value larger than zero, it will be initialized to a regular instance. To make an instance a single instance, set na or nb to 0.

This method was introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'na'. This is the setter.

nb

Signature: *[const]* unsigned long **nb**

Description: Gets the number of instances in the 'b' axis

Python specific notes:
The object exposes a readable attribute 'nb'. This is the getter.

nb=

Signature: void **nb=** (unsigned long n)

Description: Sets the number of instances in the 'b' axis

If the instance was not regular before this property is set to a value larger than zero, it will be initialized to a regular instance. To make an instance a single instance, set na or nb to 0.

This method was introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'nb'. This is the setter.

new

(1) Signature: *[static]* new CellInstArray ptr **new**

Description: Creates en empty cell instance with size 0

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new CellInstArray ptr **new** (unsigned int cell_index, const Trans trans)

Description: Creates a single cell instance

cell_index:	The cell to instantiate
trans:	The transformation by which to instantiate the cell

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new CellInstArray ptr **new** (const Cell ptr cell, const Trans trans)

Description: Creates a single cell instance

cell: The cell to instantiate

trans: The transformation by which to instantiate the cell

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new CellInstArray ptr **new** (unsigned int cell_index, const Vector disp)

Description: Creates a single cell instance

cell_index: The cell to instantiate

disp: The displacement

This convenience initializer has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new CellInstArray ptr **new** (const Cell ptr cell, const Vector disp)

Description: Creates a single cell instance

cell: The cell to instantiate

disp: The displacement

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new CellInstArray ptr **new** (unsigned int cell_index, const ICplxTrans trans)

Description: Creates a single cell instance with a complex transformation

cell_index: The cell to instantiate

trans: The complex transformation by which to instantiate the cell

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new CellInstArray ptr **new** (const Cell ptr cell, const ICplxTrans trans)

Description: Creates a single cell instance with a complex transformation

cell: The cell to instantiate

trans: The complex transformation by which to instantiate the cell

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new CellInstArray ptr **new** (unsigned int cell_index, const Trans trans, const Vector a, const Vector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance

cell_index: The cell to instantiate

trans: The transformation by which to instantiate the cell

a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis
nb:	The number of placements in the 'b' axis

Starting with version 0.25 the displacements are of vector type.

Python specific notes:
This method is the default initializer of the object.

(9) Signature: *[static]* new CellInstArray ptr **new** (const Cell ptr cell, const Trans trans, const Vector a, const Vector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance

cell:	The cell to instantiate
trans:	The transformation by which to instantiate the cell
a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis
nb:	The number of placements in the 'b' axis

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(10) Signature: *[static]* new CellInstArray ptr **new** (unsigned int cell_index, const Vector disp, const Vector a, const Vector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance

cell_index:	The cell to instantiate
disp:	The basic displacement of the first instance
a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis
nb:	The number of placements in the 'b' axis

This convenience initializer has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(11) Signature: *[static]* new CellInstArray ptr **new** (const Cell ptr cell, const Vector disp, const Vector a, const Vector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance

cell:	The cell to instantiate
disp:	The basic displacement of the first instance
a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis
nb:	The number of placements in the 'b' axis

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(12) Signature: *[static]* new CellInstArray ptr **new** (unsigned int cell_index, const ICplxTrans trans, const Vector a, const Vector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance with a complex transformation

cell_index:	The cell to instantiate
trans:	The complex transformation by which to instantiate the cell
a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis
nb:	The number of placements in the 'b' axis

Starting with version 0.25 the displacements are of vector type.

Python specific notes:
This method is the default initializer of the object.

(13) Signature: *[static]* new CellInstArray ptr **new** (const Cell ptr cell, const ICplxTrans trans, const Vector a, const Vector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance with a complex transformation

cell:	The cell to instantiate
trans:	The complex transformation by which to instantiate the cell
a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis
nb:	The number of placements in the 'b' axis

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

size

Signature: *[const]* unsigned long **size**

Description: Gets the number of single instances in the array

If the instance represents a single instance, the count is 1. Otherwise it is na*nb. Starting with version 0.27, there may be iterated instances for which the size is larger than 1, but is_regular_array? will return false. In this case, use each_trans or each_cplx_trans to retrieve the individual placements of the iterated instance.

Python specific notes:
This method is also available as 'len(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Converts the array to a string

This method was introduced in version 0.22.

Python specific notes:
This method is also available as 'str(object)'.

trans

Signature: *[const]* Trans **trans**

Description: Gets the transformation of the first instance in the array

The transformation returned is only valid if the array does not represent a complex transformation array

Python specific notes:
The object exposes a readable attribute 'trans'. This is the getter.

trans=

Signature: void **trans=** (const Trans t)

Description: Sets the transformation of the instance or the first instance in the array

This method was introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.

transform

(1) Signature: void **transform** (const Trans trans)

Description: Transforms the cell instance with the given transformation

This method has been introduced in version 0.20.

(2) Signature: void **transform** (const ICplxTrans trans)

Description: Transforms the cell instance with the given complex transformation

This method has been introduced in version 0.20.

transformed

(1) Signature: *[const]* CellInstArray **transformed** (const Trans trans)

Description: Gets the transformed cell instance

This method has been introduced in version 0.20.

(2) Signature: *[const]* CellInstArray **transformed** (const ICplxTrans trans)

Description: Gets the transformed cell instance (complex transformation)

This method has been introduced in version 0.20.

4.23. API reference - Class DCellInstArray

Notation used in Ruby API documentation

Module: db

Description: A single or array cell instance in micrometer units

This object is identical to CellInstArray, except that it holds coordinates in micron units instead of database units.

This class has been introduced in version 0.25.

Public constructors

new DCellInstArray ptr	**new**		Creates en empty cell instance with size 0
new DCellInstArray ptr	**new**	(unsigned int cell_index, const DTrans trans)	Creates a single cell instance
new DCellInstArray ptr	**new**	(const Cell ptr cell, const DTrans trans)	Creates a single cell instance
new DCellInstArray ptr	**new**	(unsigned int cell_index, const DVector disp)	Creates a single cell instance
new DCellInstArray ptr	**new**	(const Cell ptr cell, const DVector disp)	Creates a single cell instance
new DCellInstArray ptr	**new**	(unsigned int cell_index, const DCplxTrans trans)	Creates a single cell instance with a complex transformation
new DCellInstArray ptr	**new**	(const Cell ptr cell, const DCplxTrans trans)	Creates a single cell instance with a complex transformation
new DCellInstArray ptr	**new**	(unsigned int cell_index, const DTrans trans, const DVector a, const DVector b, unsigned long na, unsigned long nb)	Creates a single cell instance
new DCellInstArray ptr	**new**	(const Cell ptr cell, const DTrans trans, const DVector a, const DVector b, unsigned long na, unsigned long nb)	Creates a single cell instance
new DCellInstArray ptr	**new**	(unsigned int cell_index, const DVector disp, const DVector a, const DVector b, unsigned long na, unsigned long nb)	Creates a single cell instance
new DCellInstArray ptr	**new**	(const Cell ptr cell, const DVector disp, const DVector a, const DVector b, unsigned long na,	Creates a single cell instance

		unsigned long nb)	
new DCellInstArray ptr	new	(unsigned int cell_index, const DCplxTrans trans, const DVector a, const DVector b, unsigned long na, unsigned long nb)	Creates a single cell instance with a complex transformation
new DCellInstArray ptr	new	(const Cell ptr cell, const DCplxTrans trans, const DVector a, const DVector b, unsigned long na, unsigned long nb)	Creates a single cell instance with a complex transformation

Public methods

[const]	bool	!=	(const DCellInstArray other)	Compares two arrays for inequality
[const]	bool	<	(const DCellInstArray other)	Compares two arrays for 'less'
[const]	bool	==	(const DCellInstArray other)	Compares two arrays for equality
	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	DVector	a		Gets the displacement vector for the 'a' axis
	void	a=	(const DVector vector)	Sets the displacement vector for the 'a' axis
	void	assign	(const DCellInstArray other)	Assigns another object to self
[const]	DVector	b		Gets the displacement vector for the 'b' axis
	void	b=	(const DVector vector)	Sets the displacement vector for the 'b' axis

[const]	DBox	**bbox**	(const Layout layout, unsigned int layer_index)	Gets the bounding box of the array with respect to one layer
[const]	DBox	**bbox**	(const Layout layout)	Gets the bounding box of the array
	void	**cell=**	(Cell ptr cell)	Sets the cell this instance refers to
[const]	unsigned int	**cell_index**		Gets the cell index of the cell instantiated
	void	**cell_index=**	(unsigned int index)	Sets the index of the cell this instance refers to
[const]	DCplxTrans	**cplx_trans**		Gets the complex transformation of the first instance in the array
	void	**cplx_trans=**	(const DCplxTrans trans)	Sets the complex transformation of the instance or the first instance in the array
[const]	new DCellInstArray ptr	**dup**		Creates a copy of self
[const,iter]	DCplxTrans	**each_cplx_trans**		Gets the complex transformations represented by this instance
[const,iter]	DTrans	**each_trans**		Gets the simple transformations represented by this instance
[const]	unsigned long	**hash**		Computes a hash value
	void	**invert**		Inverts the array reference
[const]	bool	**is_complex?**		Gets a value indicating whether the array is a complex array
[const]	bool	**is_regular_array?**		Gets a value indicating whether this instance is a regular array
[const]	unsigned long	**na**		Gets the number of instances in the 'a' axis
	void	**na=**	(unsigned long n)	Sets the number of instances in the 'a' axis
[const]	unsigned long	**nb**		Gets the number of instances in the 'b' axis
	void	**nb=**	(unsigned long n)	Sets the number of instances in the 'b' axis
[const]	unsigned long	**size**		Gets the number of single instances in the array
[const]	string	**to_s**		Converts the array to a string
[const]	DTrans	**trans**		Gets the transformation of the first instance in the array

	void	**trans=**	(const DTrans t)	Sets the transformation of the instance or the first instance in the array
	void	**transform**	(const DTrans trans)	Transforms the cell instance with the given transformation
	void	**transform**	(const DCplxTrans trans)	Transforms the cell instance with the given complex transformation
[const]	DCellInstArray	**transformed**	(const DTrans trans)	Gets the transformed cell instance
[const]	DCellInstArray	**transformed**	(const DCplxTrans trans)	Gets the transformed cell instance (complex transformation)

Deprecated methods (protected, public, static, non-static and constructors)

[const]	DBox	**bbox_per_layer**	(const Layout layout, unsigned int layer_index)	Use of this method is deprecated. Use bbox instead
	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const DCellInstArray other)

Description: Compares two arrays for inequality

<

Signature: *[const]* bool **<** (const DCellInstArray other)

Description: Compares two arrays for 'less'

The comparison provides an arbitrary sorting criterion and not specific sorting order. It is guaranteed that if an array a is less than b, b is not less than a. In addition, it a is not less than b and b is not less than a, then a is equal to b.

==

Signature: *[const]* bool **==** (const DCellInstArray other)

Description: Compares two arrays for equality

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

a

Signature: *[const]* DVector **a**

Description: Gets the displacement vector for the 'a' axis

Python specific notes:
The object exposes a readable attribute 'a'. This is the getter.

a=

Signature: void **a=** (const DVector vector)

Description: Sets the displacement vector for the 'a' axis

If the instance was not regular before this property is set, it will be initialized to a regular instance.

Python specific notes:
The object exposes a writable attribute 'a'. This is the setter.

assign

Signature: void **assign** (const DCellInstArray other)

Description: Assigns another object to self

b

Signature: *[const]* DVector **b**

Description: Gets the displacement vector for the 'b' axis

Python specific notes:
The object exposes a readable attribute 'b'. This is the getter.

b=

Signature: void **b=** (const DVector vector)

Description: Sets the displacement vector for the 'b' axis

If the instance was not regular before this property is set, it will be initialized to a regular instance.

Python specific notes:
The object exposes a writable attribute 'b'. This is the setter.

bbox

(1) Signature: *[const]* DBox **bbox** (const Layout layout, unsigned int layer_index)

Description: Gets the bounding box of the array with respect to one layer

The bounding box incorporates all instances that the array represents. It needs the layout object to access the actual cell from the cell index.

'bbox' is the preferred synonym since version 0.28.

(2) Signature: *[const]* DBox **bbox** (const Layout layout)

Description: Gets the bounding box of the array

The bounding box incorporates all instances that the array represents. It needs the layout object to access the actual cell from the cell index.

bbox_per_layer

Signature: *[const]* DBox **bbox_per_layer** (const Layout layout, unsigned int layer_index)

Description: Gets the bounding box of the array with respect to one layer

Use of this method is deprecated. Use bbox instead

The bounding box incorporates all instances that the array represents. It needs the layout object to access the actual cell from the cell index.

'bbox' is the preferred synonym since version 0.28.

cell=

Signature: void **cell=** (Cell ptr cell)

Description: Sets the cell this instance refers to

This is a convenience method and equivalent to 'cell_index = cell.cell_index()'. There is no getter for the cell pointer because the CellInstArray object only knows about cell indexes.

This convenience method has been introduced in version 0.28.

Python specific notes:
The object exposes a writable attribute 'cell'. This is the setter.

cell_index

Signature: *[const]* unsigned int **cell_index**

Description: Gets the cell index of the cell instantiated

Use Layout#cell to get the Cell object from the cell index.

Python specific notes:
The object exposes a readable attribute 'cell_index'. This is the getter.

cell_index=

Signature: void **cell_index=** (unsigned int index)

Description: Sets the index of the cell this instance refers to

Python specific notes:
The object exposes a writable attribute 'cell_index'. This is the setter.

cplx_trans

Signature: *[const]* DCplxTrans **cplx_trans**

Description: Gets the complex transformation of the first instance in the array

This method is always applicable, compared to trans, since simple transformations can be expressed as complex transformations as well.

Python specific notes:
The object exposes a readable attribute 'cplx_trans'. This is the getter.

cplx_trans=

Signature: void **cplx_trans=** (const DCplxTrans trans)

Description: Sets the complex transformation of the instance or the first instance in the array

Python specific notes:
The object exposes a writable attribute 'cplx_trans'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new DCellInstArray ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_cplx_trans

Signature: *[const,iter]* DCplxTrans **each_cplx_trans**

Description: Gets the complex transformations represented by this instance

For a single instance, this iterator will deliver the single, complex transformation. For array instances, the iterator will deliver each complex transformation of the expanded array. This iterator is a generalization of each_trans for general complex transformations.

each_trans

Signature: *[const,iter]* DTrans **each_trans**

Description: Gets the simple transformations represented by this instance

For a single instance, this iterator will deliver the single, simple transformation. For array instances, the iterator will deliver each simple transformation of the expanded array.

This iterator will only deliver valid transformations if the instance array is not of complex type (see is_complex?). A more general iterator that delivers the complex transformations is each_cplx_trans.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given cell instance. This method enables cell instances as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

invert

Signature: void **invert**

Description: Inverts the array reference

The inverted array reference describes in which transformations the parent cell is seen from the current cell.

is_complex?

Signature: *[const]* bool **is_complex?**

Description: Gets a value indicating whether the array is a complex array

Returns true if the array represents complex instances (that is, with magnification and arbitrary rotation angles).

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_regular_array?

Signature: *[const]* bool **is_regular_array?**

Description: Gets a value indicating whether this instance is a regular array

na

Signature: *[const]* unsigned long **na**

Description: Gets the number of instances in the 'a' axis

Python specific notes:
The object exposes a readable attribute 'na'. This is the getter.

na=

Signature: void **na=** (unsigned long n)

Description: Sets the number of instances in the 'a' axis

If the instance was not regular before this property is set to a value larger than zero, it will be initialized to a regular instance. To make an instance a single instance, set na or nb to 0.

Python specific notes:
The object exposes a writable attribute 'na'. This is the setter.

nb

Signature: *[const]* unsigned long **nb**

Description: Gets the number of instances in the 'b' axis

Python specific notes:
The object exposes a readable attribute 'nb'. This is the getter.

nb=

Signature: void **nb=** (unsigned long n)

Description: Sets the number of instances in the 'b' axis

If the instance was not regular before this property is set to a value larger than zero, it will be initialized to a regular instance. To make an instance a single instance, set na or nb to 0.

Python specific notes:
The object exposes a writable attribute 'nb'. This is the setter.

new

(1) Signature: *[static]* new DCellInstArray ptr **new**

Description: Creates en empty cell instance with size 0

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DCellInstArray ptr **new** (unsigned int cell_index, const DTrans trans)

Description: Creates a single cell instance

cell_index:	The cell to instantiate
trans:	The transformation by which to instantiate the cell

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new DCellInstArray ptr **new** (const Cell ptr cell, const DTrans trans)

Description: Creates a single cell instance

cell:	The cell to instantiate
trans:	The transformation by which to instantiate the cell

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new DCellInstArray ptr **new** (unsigned int cell_index, const DVector disp)

Description: Creates a single cell instance

cell_index:	The cell to instantiate
disp:	The displacement

This convenience initializer has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new DCellInstArray ptr **new** (const Cell ptr cell, const DVector disp)

Description: Creates a single cell instance

cell:	The cell to instantiate
disp:	The displacement

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new DCellInstArray ptr **new** (unsigned int cell_index, const DCplxTrans trans)

Description: Creates a single cell instance with a complex transformation

cell_index:	The cell to instantiate
trans:	The complex transformation by which to instantiate the cell

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new DCellInstArray ptr **new** (const Cell ptr cell, const DCplxTrans trans)

Description: Creates a single cell instance with a complex transformation

cell:	The cell to instantiate
trans:	The complex transformation by which to instantiate the cell

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new DCellInstArray ptr **new** (unsigned int cell_index, const DTrans trans, const DVector a, const DVector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance

cell_index:	The cell to instantiate
trans:	The transformation by which to instantiate the cell
a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis
nb:	The number of placements in the 'b' axis

Python specific notes:
This method is the default initializer of the object.

(9) Signature: *[static]* new DCellInstArray ptr **new** (const Cell ptr cell, const DTrans trans, const DVector a, const DVector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance

cell:	The cell to instantiate

trans:	The transformation by which to instantiate the cell
a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis
nb:	The number of placements in the 'b' axis

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(10) Signature: *[static]* new DCellInstArray ptr **new** (unsigned int cell_index, const DVector disp, const DVector a, const DVector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance

cell_index:	The cell to instantiate
disp:	The basic displacement of the first instance
a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis
nb:	The number of placements in the 'b' axis

This convenience initializer has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(11) Signature: *[static]* new DCellInstArray ptr **new** (const Cell ptr cell, const DVector disp, const DVector a, const DVector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance

cell:	The cell to instantiate
disp:	The basic displacement of the first instance
a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis
nb:	The number of placements in the 'b' axis

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

(12) Signature: *[static]* new DCellInstArray ptr **new** (unsigned int cell_index, const DCplxTrans trans, const DVector a, const DVector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance with a complex transformation

cell_index:	The cell to instantiate
trans:	The complex transformation by which to instantiate the cell
a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis

nb:	The number of placements in the 'b' axis

Python specific notes:
This method is the default initializer of the object.

(13) Signature: *[static]* new DCellInstArray ptr **new** (const Cell ptr cell, const DCplxTrans trans, const DVector a, const DVector b, unsigned long na, unsigned long nb)

Description: Creates a single cell instance with a complex transformation

cell:	The cell to instantiate
trans:	The complex transformation by which to instantiate the cell
a:	The displacement vector of the array in the 'a' axis
b:	The displacement vector of the array in the 'b' axis
na:	The number of placements in the 'a' axis
nb:	The number of placements in the 'b' axis

This convenience variant takes a Cell pointer and is equivalent to using 'cell.cell_index()'. It has been introduced in version 0.28.

Python specific notes:
This method is the default initializer of the object.

size

Signature: *[const]* unsigned long **size**

Description: Gets the number of single instances in the array

If the instance represents a single instance, the count is 1. Otherwise it is na*nb. Starting with version 0.27, there may be iterated instances for which the size is larger than 1, but is_regular_array? will return false. In this case, use each_trans or each_cplx_trans to retrieve the individual placements of the iterated instance.

Python specific notes:
This method is also available as 'len(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Converts the array to a string

Python specific notes:
This method is also available as 'str(object)'.

trans

Signature: *[const]* DTrans **trans**

Description: Gets the transformation of the first instance in the array

The transformation returned is only valid if the array does not represent a complex transformation array

Python specific notes:
The object exposes a readable attribute 'trans'. This is the getter.

trans=

Signature: void **trans=** (const DTrans t)

Description: Sets the transformation of the instance or the first instance in the array

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.

transform

(1) Signature: void **transform** (const DTrans trans)

Description: Transforms the cell instance with the given transformation

	(2) Signature: void **transform** (const DCplxTrans trans) **Description**: Transforms the cell instance with the given complex transformation
transformed	**(1) Signature**: *[const]* DCellInstArray **transformed** (const DTrans trans) **Description**: Gets the transformed cell instance **(2) Signature**: *[const]* DCellInstArray **transformed** (const DCplxTrans trans) **Description**: Gets the transformed cell instance (complex transformation)

4.24. API reference - Class CellMapping

Notation used in Ruby API documentation

Module: db

Description: A cell mapping (source to target layout)

A cell mapping is an association of cells in two layouts forming pairs of cells, i.e. one cell corresponds to another cell in the other layout. The CellMapping object describes the mapping of cells of a source layout B to a target layout A. The cell mapping object is basically a table associating a cell in layout B with a cell in layout A.

The cell mapping is of particular interest for providing the cell mapping recipe in Cell#copy_tree_shapes or Cell#move_tree_shapes.

The mapping object is used to create and hold that table. There are three basic modes in which a table can be generated:

- Top-level identity (for_single_cell and for_single_cell_full)
- Top-level identify for multiple cells (for_multi_cells_full and for_multi_cells_full)
- Geometrical identity (from_geometry and from_geometry_full)
- Name identity (from_names and from_names_full)

'full' refers to the way cells are treated which are not mentioned. In the 'full' versions, cells for which no mapping is established explicitly - specifically all child cells in top-level identity modes - are created in the target layout and instantiated according to their source layout hierarchy. Then, these new cells become targets of the respective source cells. In the plain version (without 'full' cells), no additional cells are created. For the case of Layout#copy_tree_shapes cells not explicitly mapped are flattened. Hence for example, for_single_cell will flatten all children of the source cell during Layout#copy_tree_shapes or Layout#move_tree_shapes.

Top-level identity means that only one cell (the top cell) is regarded identical. All child cells are not considered identical. In full mode (see below), this will create a new, identical cell tree below the top cell in layout A.

Geometrical identity is defined by the exact identity of the set of expanded instances in each starting cell. Therefore, when a cell is mapped to another cell, shapes can be transferred from one cell to another while effectively rendering the same flat geometry (in the context of the given starting cells). Location identity is basically the safest way to map cells from one hierarchy into another, because it preserves the flat shape geometry. However in some cases the algorithm may find multiple mapping candidates. In that case it will make a guess about what mapping to choose.

Name identity means that cells are identified by their names - for a source cell in layer B, a target cell with the same name is looked up in the target layout A and a mapping is created if a cell with the same name is found. However, name identity does not mean that the cells are actually equivalent because they may be placed differently. Hence, cell mapping by name is not a good choice when it is important to preserve the shape geometry of a layer.

A cell might not be mapped to another cell which basically means that there is no corresponding cell. In this case, flattening to the next mapped cell is an option to transfer geometries despite the missing mapping. You can enforce a mapping by using the mapping generator methods in 'full' mode, i.e. from_names_full or from_geometry_full. These versions will create new cells and their corresponding instances in the target layout if no suitable target cell is found.

This is a simple example for a cell mapping preserving the hierarchy of the source cell and creating a hierarchy copy in the top cell of the target layout ('hierarchical merge'):

```
cell_names = [ "A", "B", "C" ]

source = RBA::Layout::new
source.read("input.gds")

target = RBA::Layout::new
target_top = target.create_cell("IMPORTED")

cm = RBA::CellMapping::new
# Copies the source layout hierarchy into the target top cell:
cm.for_single_cell_full(target_top, source.top_cell)
target.copy_tree_shapes(source, cm)
```

Without 'full', the effect is move-with-flattening (note we're using 'move' in this example):

```
cell_names = [ "A", "B", "C" ]

source = RBA::Layout::new
source.read("input.gds")

target = RBA::Layout::new
target_top = target.create_cell("IMPORTED")

cm = RBA::CellMapping::new
# Flattens the source layout hierarchy into the target top cell:
cm.for_single_cell(target_top, source.top_cell)
target.move_tree_shapes(source, cm)
```

This is another example for using CellMapping in multiple top cell identity mode. It extracts cells 'A', 'B' and 'C' from one layout and copies them to another. It will also copy all shapes and all child cells. Child cells which are shared between the three initial cells will be shared in the target layout too.

```
cell_names = [ "A", "B", "C" ]

source = RBA::Layout::new
source.read("input.gds")

target = RBA::Layout::new

source_cells = cell_names.collect { |n| source.cell_by_name(n) }
target_cells = cell_names.collect { |n| target.create_cell(n) }

cm = RBA::CellMapping::new
cm.for_multi_cells_full(target_cells, source_cells)
target.copy_tree_shapes(source, cm)
```

Public constructors

new CellMapping ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const CellMapping other)	Assigns another object to self

[const]	unsigned int	**cell_mapping**	(unsigned int cell_index_b)	Determines cell mapping of a layout_b cell to the corresponding layout_a cell.
	void	**clear**		Clears the mapping.
[const]	new CellMapping ptr	**dup**		Creates a copy of self
	void	**for_multi_cells**	(const Layout layout_a, unsigned int[] cell_indexes_a, const Layout layout_b, unsigned int[] cell_indexes_b)	Initializes the cell mapping for top-level identity
	void	**for_multi_cells**	(Cell ptr[] cell_a, const Cell ptr[] cell_b)	Initializes the cell mapping for top-level identity
	unsigned int[]	**for_multi_cells_full**	(Layout layout_a, unsigned int[] cell_indexes_a, const Layout layout_b, unsigned int[] cell_indexes_b)	Initializes the cell mapping for top-level identity in full mapping mode
	unsigned int[]	**for_multi_cells_full**	(Cell ptr[] cell_a, const Cell ptr[] cell_b)	Initializes the cell mapping for top-level identity in full mapping mode
	void	**for_single_cell**	(const Layout layout_a, unsigned int cell_index_a, const Layout layout_b, unsigned int cell_index_b)	Initializes the cell mapping for top-level identity
	void	**for_single_cell**	(Cell cell_a, const Cell cell_b)	Initializes the cell mapping for top-level identity
	unsigned int[]	**for_single_cell_full**	(Layout layout_a, unsigned int cell_index_a, const Layout layout_b, unsigned int cell_index_b)	Initializes the cell mapping for top-level identity in full mapping mode
	unsigned int[]	**for_single_cell_full**	(Cell cell_a, const Cell cell_b)	Initializes the cell mapping for top-level identity in full mapping mode
	void	**from_geometry**	(const Layout layout_a, unsigned int cell_index_a, const Layout layout_b, unsigned int cell_index_b)	Initializes the cell mapping using the geometrical identity
	void	**from_geometry**	(Cell cell_a, const Cell cell_b)	Initializes the cell mapping using the geometrical identity
	unsigned int[]	**from_geometry_full**	(Layout layout_a, unsigned int cell_index_a, const Layout layout_b, unsigned int cell_index_b)	Initializes the cell mapping using the geometrical identity in full mapping mode
	unsigned int[]	**from_geometry_full**	(Cell cell_a, const Cell cell_b)	Initializes the cell mapping using the geometrical identity in full mapping mode
	void	**from_names**	(const Layout layout_a, unsigned int cell_index_a, const Layout layout_b,	Initializes the cell mapping using the name identity

			unsigned int cell_index_b)	
	void	**from_names**	(Cell cell_a, const Cell cell_b)	Initializes the cell mapping using the name identity
	unsigned int[]	**from_names_full**	(Layout layout_a, unsigned int cell_index_a, const Layout layout_b, unsigned int cell_index_b)	Initializes the cell mapping using the name identity in full mapping mode
	unsigned int[]	**from_names_full**	(Cell cell_a, const Cell cell_b)	Initializes the cell mapping using the name identity in full mapping mode
[const]	bool	**has_mapping?**	(unsigned int cell_index_b)	Returns as value indicating whether a cell of layout_b has a mapping to a layout_a cell.
	void	**map**	(unsigned int cell_index_b, unsigned int cell_index_a)	Explicitly specifies a mapping.
[const]	map<unsigned int,unsigned int>	**table**		Returns the mapping table.

Public static methods and constants

unsigned int	**DropCell**	A constant indicating the request to drop a cell

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

DropCell

Signature: *[static]* unsigned int **DropCell**

Description: A constant indicating the request to drop a cell

If used as a pseudo-target for the cell mapping, this index indicates that the cell shall be dropped rather than created on the target side or skipped by flattening. Instead, all shapes of this cell are discarded and its children are not translated unless explicitly requested or if required are children for other cells.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'DropCell'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const CellMapping other)

Description: Assigns another object to self

cell_mapping

Signature: *[const]* unsigned int **cell_mapping** (unsigned int cell_index_b)

Description: Determines cell mapping of a layout_b cell to the corresponding layout_a cell.

cell_index_b:	The index of the cell in layout_b whose mapping is requested.
Returns:	The cell index in layout_a.

Note that the returned index can be DropCell to indicate the cell shall be dropped.

clear

Signature: void **clear**

Description: Clears the mapping.

This method has been introduced in version 0.23.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new CellMapping ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

for_multi_cells

(1) Signature: void **for_multi_cells** (const Layout layout_a, unsigned int[] cell_indexes_a, const Layout layout_b, unsigned int[] cell_indexes_b)

Description: Initializes the cell mapping for top-level identity

layout_a:	The target layout.
cell_indexes_a:	A list of cell indexes for the target cells.
layout_b:	The source layout.
cell_indexes_b:	A list of cell indexes for the source cells (same number of indexes than cell_indexes_a).

The cell mapping is created for cells from cell_indexes_b to cell from cell_indexes_a in the respective layouts. This method clears the mapping and creates one for each cell pair from cell_indexes_b vs. cell_indexes_a. If used for Layout#copy_tree_shapes or Layout#move_tree_shapes, this cell mapping will essentially flatten the source cells in the target layout.

This method is equivalent to clear, followed by map(cell_index_a, cell_index_b) for each cell pair.

This method has been introduced in version 0.27.

(2) Signature: void **for_multi_cells** (Cell ptr[] cell_a, const Cell ptr[] cell_b)

Description: Initializes the cell mapping for top-level identity

cell_a: A list of target cells.

cell_b: A list of source cells.

Returns: A list of indexes of cells created.

This is a convenience version which uses cell references instead of layout/cell index combinations. It has been introduced in version 0.28.

for_multi_cells_full

(1) Signature: unsigned int[] **for_multi_cells_full** (Layout layout_a, unsigned int[] cell_indexes_a, const Layout layout_b, unsigned int[] cell_indexes_b)

Description: Initializes the cell mapping for top-level identity in full mapping mode

layout_a: The target layout.

cell_indexes_a: A list of cell indexes for the target cells.

layout_b: The source layout.

cell_indexes_b: A list of cell indexes for the source cells (same number of indexes than cell_indexes_a).

The cell mapping is created for cells from cell_indexes_b to cell from cell_indexes_a in the respective layouts. This method clears the mapping and creates one for each cell pair from cell_indexes_b vs. cell_indexes_a. In addition and in contrast to for_multi_cells, this method completes the mapping by adding all the child cells of all cells in cell_indexes_b to layout_a and creating the proper instances.

This method has been introduced in version 0.27.

(2) Signature: unsigned int[] **for_multi_cells_full** (Cell ptr[] cell_a, const Cell ptr[] cell_b)

Description: Initializes the cell mapping for top-level identity in full mapping mode

cell_a: A list of target cells.

cell_b: A list of source cells.

Returns: A list of indexes of cells created.

This is a convenience version which uses cell references instead of layout/cell index combinations. It has been introduced in version 0.28.

for_single_cell

(1) Signature: void **for_single_cell** (const Layout layout_a, unsigned int cell_index_a, const Layout layout_b, unsigned int cell_index_b)

Description: Initializes the cell mapping for top-level identity

layout_a: The target layout.

cell_index_a: The index of the target cell.

layout_b: The source layout.

cell_index_b: The index of the source cell.

The cell mapping is created for cell_b to cell_a in the respective layouts. This method clears the mapping and creates one for the single cell pair. If used for Cell#copy_tree or Cell#move_tree, this cell mapping will essentially flatten the cell.

This method is equivalent to clear, followed by map(cell_index_a, cell_index_b).

This method has been introduced in version 0.23.

(2) Signature: void **for_single_cell** (Cell cell_a, const Cell cell_b)

Description: Initializes the cell mapping for top-level identity

cell_a: The target cell.

cell_b: The source cell.

Returns: A list of indexes of cells created.

This is a convenience version which uses cell references instead of layout/cell index combinations. It has been introduced in version 0.28.

for_single_cell_full

(1) Signature: unsigned int[] **for_single_cell_full** (Layout layout_a, unsigned int cell_index_a, const Layout layout_b, unsigned int cell_index_b)

Description: Initializes the cell mapping for top-level identity in full mapping mode

layout_a: The target layout.

cell_index_a: The index of the target cell.

layout_b: The source layout.

cell_index_b: The index of the source cell.

The cell mapping is created for cell_b to cell_a in the respective layouts. This method clears the mapping and creates one for the single cell pair. In addition and in contrast to for_single_cell, this method completes the mapping by adding all the child cells of cell_b to layout_a and creating the proper instances.

This method has been introduced in version 0.23.

(2) Signature: unsigned int[] **for_single_cell_full** (Cell cell_a, const Cell cell_b)

Description: Initializes the cell mapping for top-level identity in full mapping mode

cell_a: The target cell.

cell_b: The source cell.

Returns: A list of indexes of cells created.

This is a convenience version which uses cell references instead of layout/cell index combinations. It has been introduced in version 0.28.

from_geometry

(1) Signature: void **from_geometry** (const Layout layout_a, unsigned int cell_index_a, const Layout layout_b, unsigned int cell_index_b)

Description: Initializes the cell mapping using the geometrical identity

layout_a: The target layout.

cell_index_a: The index of the target starting cell.

layout_b: The source layout.

cell_index_b: The index of the source starting cell.

The cell mapping is created for cells below cell_a and cell_b in the respective layouts. This method employs geometrical identity to derive mappings for the child cells of the starting cell in layout A and B. If the geometrical identity is ambiguous, the algorithm will make an arbitrary choice.

This method has been introduced in version 0.23.

(2) Signature: void **from_geometry** (Cell cell_a, const Cell cell_b)

Description: Initializes the cell mapping using the geometrical identity

cell_a: The target cell.

cell_b: The source cell.

Returns: A list of indexes of cells created.

This is a convenience version which uses cell references instead of layout/cell index combinations. It has been introduced in version 0.28.

from_geometry_full

(1) Signature: unsigned int[] **from_geometry_full** (Layout layout_a, unsigned int cell_index_a, const Layout layout_b, unsigned int cell_index_b)

Description: Initializes the cell mapping using the geometrical identity in full mapping mode

layout_a:	The target layout.
cell_index_a:	The index of the target starting cell.
layout_b:	The source layout.
cell_index_b:	The index of the source starting cell.
Returns:	A list of indexes of cells created.

The cell mapping is created for cells below cell_a and cell_b in the respective layouts. This method employs geometrical identity to derive mappings for the child cells of the starting cell in layout A and B. If the geometrical identity is ambiguous, the algorithm will make an arbitrary choice.

Full mapping means that cells which are not found in the target layout A are created there plus their corresponding instances are created as well. The returned list will contain the indexes of all cells created for that reason.

This method has been introduced in version 0.23.

(2) Signature: unsigned int[] **from_geometry_full** (Cell cell_a, const Cell cell_b)

Description: Initializes the cell mapping using the geometrical identity in full mapping mode

cell_a:	The target cell.
cell_b:	The source cell.
Returns:	A list of indexes of cells created.

This is a convenience version which uses cell references instead of layout/cell index combinations. It has been introduced in version 0.28.

from_names

(1) Signature: void **from_names** (const Layout layout_a, unsigned int cell_index_a, const Layout layout_b, unsigned int cell_index_b)

Description: Initializes the cell mapping using the name identity

layout_a:	The target layout.
cell_index_a:	The index of the target starting cell.
layout_b:	The source layout.
cell_index_b:	The index of the source starting cell.

The cell mapping is created for cells below cell_a and cell_b in the respective layouts. This method employs name identity to derive mappings for the child cells of the starting cell in layout A and B.

This method has been introduced in version 0.23.

(2) Signature: void **from_names** (Cell cell_a, const Cell cell_b)

Description: Initializes the cell mapping using the name identity

cell_a:	The target cell.
cell_b:	The source cell.
Returns:	A list of indexes of cells created.

This is a convenience version which uses cell references instead of layout/cell index combinations. It has been introduced in version 0.28.

from_names_full

(1) Signature: unsigned int[] **from_names_full** (Layout layout_a, unsigned int cell_index_a, const Layout layout_b, unsigned int cell_index_b)

Description: Initializes the cell mapping using the name identity in full mapping mode

layout_a:	The target layout.
cell_index_a:	The index of the target starting cell.
layout_b:	The source layout.
cell_index_b:	The index of the source starting cell.
Returns:	A list of indexes of cells created.

The cell mapping is created for cells below cell_a and cell_b in the respective layouts. This method employs name identity to derive mappings for the child cells of the starting cell in layout A and B.

Full mapping means that cells which are not found in the target layout A are created there plus their corresponding instances are created as well. The returned list will contain the indexes of all cells created for that reason.

This method has been introduced in version 0.23.

(2) Signature: unsigned int[] **from_names_full** (Cell cell_a, const Cell cell_b)

Description: Initializes the cell mapping using the name identity in full mapping mode

cell_a:	The target cell.
cell_b:	The source cell.
Returns:	A list of indexes of cells created.

This is a convenience version which uses cell references instead of layout/cell index combinations. It has been introduced in version 0.28.

has_mapping?

Signature: *[const]* bool **has_mapping?** (unsigned int cell_index_b)

Description: Returns as value indicating whether a cell of layout_b has a mapping to a layout_a cell.

cell_index_b:	The index of the cell in layout_b whose mapping is requested.
Returns:	true, if the cell has a mapping

Note that if the cell is supposed to be dropped (see DropCell), the respective source cell will also be regarded "mapped", so has_mapping? will return true in this case.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

map

Signature: void **map** (unsigned int cell_index_b, unsigned int cell_index_a)

Description: Explicitly specifies a mapping.

cell_index_b:	The index of the cell in layout B (the "source")
cell_index_a:	The index of the cell in layout A (the "target") - this index can be DropCell

Beside using the mapping generator algorithms provided through from_names and from_geometry, it is possible to explicitly specify cell mappings using this method.

This method has been introduced in version 0.23.

new

Signature: *[static]* new CellMapping ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

table

Signature: *[const]* map<unsigned int,unsigned int> **table**

Description: Returns the mapping table.

The mapping table is a dictionary where the keys are source layout cell indexes and the values are the target layout cell indexes. Note that the target cell index can be DropCell to indicate that a cell is supposed to be dropped.

This method has been introduced in version 0.25.

4.25. API reference - Class CompoundRegionOperationNode

Notation used in Ruby API documentation

Module: db

Description: A base class for compound DRC operations

Sub-classes: LogicalOp, GeometricalOp, ResultType, ParameterType, RatioParameterType

This class is not intended to be used directly but rather provide a factory for various incarnations of compound operation nodes. Compound operations are a way to specify complex DRC operations put together by building a tree of operations. This operation tree then is executed with Region#complex_op and will act on individual clusters of shapes and their interacting neighbors.

A basic concept to the compound operations is the 'subject' (primary) and 'intruder' (secondary) input. The 'subject' is the Region, 'complex_op' with the operation tree is executed on. 'intruders' are regions inserted into the equation through secondary input nodes created with new_secondary_node. The algorithm will execute the operation tree for every subject shape considering intruder shapes from the secondary inputs. The algorithm will only act on subject shapes primarily. As a consequence, 'lonely' intruder shapes without a subject shape are not considered at all. Only subject shapes trigger evaluation of the operation tree.

The search distance for intruder shapes is determined by the operation and computed from the operation's requirements.

This class has been introduced in version 0.27. The API is considered internal and will change without notice.

Public constructors

new CompoundRegionOperationNode ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	string	**description**		Gets the description for this node
	void	**description=**	(string d)	Sets the description for this node
[const]	int	**distance**		Gets the distance value for this node
	void	**distance=**	(int d)	Sets the distance value for this nodeUsually it's not required to provide a distance because the nodes compute a distance based on their operation. If necessary you can supply a distance. The processor will use this distance or the computed one, whichever is larger.
[const]	CompoundRegionOpe	**result_type**		Gets the result type of this node

Public static methods and constants

new CompoundRegionOperat ptr	new_area_filter	(CompoundRegionOperationNode ptr input, bool inverse = false, long amin = 0, long amax = max)	Creates a node filtering the input by area.
new CompoundRegionOperationNode ptr	new_area_sum_filter	(CompoundRegionOperationNode ptr input, bool inverse = false, long amin = 0, long amax = max)	Creates a node filtering the input by area sum.
new CompoundRegionOperat ptr	new_bbox_filter	(CompoundRegionOperationNode ptr input, CompoundRegionOperationNode::Parame parameter, bool inverse = false, unsigned int pmin = 0, unsigned int pmax = max)	Creates a node filtering the input by bounding box parameters.
new CompoundRegionOperationNode ptr	new_case	(CompoundRegionOperationNode ptr[] inputs)	Creates a 'switch ladder' (case statement) compound operation node.
new CompoundRegionOperat ptr	new_centers	(CompoundRegionOperationNode ptr input, unsigned int length, double fraction)	Creates a node delivering a part at the center of each input edge.
new CompoundRegionOperationNode ptr	new_convex_decomposition	(CompoundRegionOperationNode ptr input, PreferredOrientation mode)	Creates a node providing a composition into convex pieces.
new CompoundRegionOperat ptr	new_corners_as_d	(CompoundRegionOperationNode ptr input, double angle_min, bool include_angle_min, double angle_max, bool include_angle_max)	Creates a node turning corners into dots (single-point edges).
new CompoundRegionOperationNode ptr	new_corners_as_edge_pairs	(CompoundRegionOperationNode ptr input, double angle_min, bool include_angle_min, double angle_max, bool include_angle_max)	Creates a node turning corners into edge pairs containing the two edges adjacent to the corner.
new CompoundRegionOperat ptr	new_corners_as_r	(CompoundRegionOperationNode ptr input, double angle_min, bool include_angle_min, double angle_max, bool include_angle_max, int dim)	Creates a node turning corners into rectangles.

new CompoundRegionOperationNode ptr	new_count_filter	(CompoundRegionOperationNode ptr inputs, bool invert = false, unsigned long min_count = 0, unsigned long max_count = 18446744073709551615)	Creates a node selecting results but their shape count.
new CompoundRegionOperat ptr	new_edge_length_	(CompoundRegionOperationNode ptr input, bool inverse = false, unsigned int lmin = 0, unsigned int lmax = max)	Creates a node filtering edges by their length.
new CompoundRegionOperationNode ptr	new_edge_length_sum_filter	(CompoundRegionOperationNode ptr input, bool inverse = false, unsigned int lmin = 0, unsigned int lmax = max)	Creates a node filtering edges by their length sum (over the local set).
new CompoundRegionOperat ptr	new_edge_orientat	(CompoundRegionOperationNode ptr input, bool inverse, double amin, bool include_amin, double amax, bool include_amax)	Creates a node filtering edges by their orientation.
new CompoundRegionOperationNode ptr	new_edge_pair_to_first_edges	(CompoundRegionOperationNode ptr input)	Creates a node delivering the first edge of each edges pair.
new CompoundRegionOperat ptr	new_edge_pair_to_	(CompoundRegionOperationNode ptr input)	Creates a node delivering the second edge of each edges pair.
new CompoundRegionOperationNode ptr	new_edges	(CompoundRegionOperationNode ptr input, EdgeMode mode = All)	Creates a node converting polygons into its edges.
new CompoundRegionOperat ptr	new_empty	(CompoundRegionOperationNode::Result type)	Creates a node delivering an empty result of the given type
new CompoundRegionOperationNode ptr	new_enclosed_check	(CompoundRegionOperationNode ptr other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)	Creates a node providing an enclosed (secondary enclosing primary) check.

new CompoundRegionOperat ptr	new_enclosing	(CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b, bool inverse = false, unsigned long min_count = 1, unsigned long max_count = unlimited)	Creates a node representing an inside selection operation between the inputs.
new CompoundRegionOperationNode ptr	new_enclosing_check	(CompoundRegionOperationNode ptr other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)	Creates a node providing an inside (enclosure) check.
new CompoundRegionOperat ptr	new_end_segment	(CompoundRegionOperationNode ptr input, unsigned int length, double fraction)	Creates a node delivering a part at the end of each input edge.
new CompoundRegionOperationNode ptr	new_extended	(CompoundRegionOperationNode ptr input, int ext_b, int ext_e, int ext_o, int ext_i)	Creates a node delivering a polygonized version of the edges with the four extension parameters.
new CompoundRegionOperat ptr	new_extended_in	(CompoundRegionOperationNode ptr input, int e)	Creates a node delivering a polygonized, inside-extended version of the edges.
new CompoundRegionOperationNode ptr	new_extended_out	(CompoundRegionOperationNode ptr input, int e)	Creates a node delivering a polygonized, inside-extended version of the edges.
new CompoundRegionOperat ptr	new_extents	(CompoundRegionOperationNode ptr input, int e = 0)	Creates a node returning the extents of the objects.
new CompoundRegionOperationNode ptr	new_foreign		Creates a node object representing the primary input without the current polygon
new CompoundRegionOperat ptr	new_geometrical_b	(CompoundRegionOperationNode::Geome op, CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b)	Creates a node representing a geometrical boolean operation between the inputs.
new CompoundRegionOperationNode ptr	new_hole_count_filter	(CompoundRegionOperationNode ptr input, bool inverse = false,	Creates a node filtering the input by number of holes per polygon.

		unsigned long hmin = 0, unsigned long hmax = max)	
new CompoundRegionOperat ptr	new_holes	(CompoundRegionOperationNode ptr input)	Creates a node extracting the holes from polygons.
new CompoundRegionOperationNode ptr	new_hulls	(CompoundRegionOperationNode ptr input)	Creates a node extracting the hulls from polygons.
new CompoundRegionOperat ptr	new_inside	(CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b, bool inverse = false)	Creates a node representing an inside selection operation between the inputs.
new CompoundRegionOperationNode ptr	new_interacting	(CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b, bool inverse = false, unsigned long min_count = 1, unsigned long max_count = unlimited)	Creates a node representing an interacting selection operation between the inputs.
new CompoundRegionOperat ptr	new_isolated_chec	(int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)	Creates a node providing a isolated polygons (space between different polygons) check.
new CompoundRegionOperationNode ptr	new_join	(CompoundRegionOperationNode ptr[] inputs)	Creates a node that joins the inputs.
new CompoundRegionOperat ptr	new_logical_boole	(CompoundRegionOperationNode::Logical op, bool invert, CompoundRegionOperationNode ptr[] inputs)	Creates a node representing a logical boolean operation between the inputs.
new CompoundRegionOperationNode ptr	new_merged	(CompoundRegionOperationNode ptr input, bool min_coherence = false, unsigned int min_wc = 0)	Creates a node providing merged input polygons.
new CompoundRegionOperat ptr	new_minkowski_su	(CompoundRegionOperationNode ptr input, const Edge e)	Creates a node providing a Minkowski sum with an edge.
new CompoundRegionOperationNode ptr	new_minkowski_sum	(CompoundRegionOperationNode ptr input, const Polygon p)	Creates a node providing a Minkowski sum with a polygon.

new CompoundRegionOperat ptr	new_minkowski_su	(CompoundRegionOperationNode ptr input, const Box p)	Creates a node providing a Minkowski sum with a box.
new CompoundRegionOperationNode ptr	new_minkowski_sum	(CompoundRegionOperationNode ptr input, Point[] p)	Creates a node providing a Minkowski sum with a point sequence forming a contour.
new CompoundRegionOperat ptr	new_notch_check	(int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)	Creates a node providing a intra-polygon space check.
new CompoundRegionOperationNode ptr	new_outside	(CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b, bool inverse = false)	Creates a node representing an outside selection operation between the inputs.
new CompoundRegionOperat ptr	new_overlap_chec	(CompoundRegionOperationNode ptr other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)	Creates a node providing an overlap check.
new CompoundRegionOperationNode ptr	new_overlapping	(CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b, bool inverse = false, unsigned long min_count = 1, unsigned long max_count = unlimited)	Creates a node representing an overlapping selection operation between the inputs.
new CompoundRegionOperat ptr	new_perimeter_filt	(CompoundRegionOperationNode ptr input, bool inverse = false, unsigned long pmin = 0, unsigned long pmax = max)	Creates a node filtering the input by perimeter.
new CompoundRegionOperationNode ptr	new_perimeter_sum_filter	(CompoundRegionOperationNode ptr input, bool inverse = false, unsigned long amin = 0, unsigned long amax = max)	Creates a node filtering the input by area sum.

new CompoundRegionOperat ptr	new_polygon_brea	(CompoundRegionOperationNode ptr input, unsigned long max_vertex_count, double max_area_ratio)	Creates a node providing a composition into parts with less than the given number of points and a smaller area ratio.
new CompoundRegionOperationNode ptr	new_polygons	(CompoundRegionOperationNode ptr input, int e = 0)	Creates a node converting the input to polygons.
new CompoundRegionOperat ptr	new_primary		Creates a node object representing the primary input
new CompoundRegionOperationNode ptr	new_ratio_filter	(CompoundRegionOperationNode ptr input, CompoundRegionOperationNode::RatioParameterType parameter, bool inverse = false, double pmin = 0, bool pmin_included = true, double pmax = max, bool pmax_included = true)	Creates a node filtering the input by ratio parameters.
new CompoundRegionOperat ptr	new_rectangle_filt	(CompoundRegionOperationNode ptr input, bool is_square = false, bool inverse = false)	Creates a node filtering the input for rectangular or square shapes.
new CompoundRegionOperationNode ptr	new_rectilinear_filter	(CompoundRegionOperationNode ptr input, bool inverse = false)	Creates a node filtering the input for rectilinear shapes (or non-rectilinear ones with 'inverse' set to 'true').
new CompoundRegionOperat ptr	new_relative_exter	(CompoundRegionOperationNode ptr input, double fx1, double fy1, double fx2, double fy2, int dx, int dy)	Creates a node returning markers at specified locations of the extent (e.g. at the center).
new CompoundRegionOperationNode ptr	new_relative_extents_as_edges	(CompoundRegionOperationNode ptr input, double fx1, double fy1, double fx2, double fy2)	Creates a node returning edges at specified locations of the extent (e.g. at the center).
new CompoundRegionOperat ptr	new_rounded_corr	(CompoundRegionOperationNode ptr input, double rinner, double router, unsigned int n)	Creates a node generating rounded corners.
new CompoundRegionOperationNode ptr	new_secondary	(Region ptr region)	Creates a node object representing the secondary input from the given region

new CompoundRegionOperat ptr	new_separation_cl	(CompoundRegionOperationNode ptr other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)	Creates a node providing a separation check.
new CompoundRegionOperationNode ptr	new_sized	(CompoundRegionOperationNode ptr input, int dx, int dy, unsigned int mode)	Creates a node providing sizing.
new CompoundRegionOperat ptr	new_smoothed	(CompoundRegionOperationNode ptr input, int d, bool keep_hv = false)	Creates a node smoothing the polygons.
new CompoundRegionOperationNode ptr	new_space_check	(int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)	Creates a node providing a space check.
new CompoundRegionOperat ptr	new_start_segmen	(CompoundRegionOperationNode ptr input, unsigned int length, double fraction)	Creates a node delivering a part at the beginning of each input edge.
new CompoundRegionOperationNode ptr	new_strange_polygon_filter	(CompoundRegionOperationNode ptr input)	Creates a node extracting strange polygons.
new CompoundRegionOperat ptr	new_trapezoid_dec	(CompoundRegionOperationNode ptr input, TrapezoidDecompositionMode mode)	Creates a node providing a composition into trapezoids.

	new CompoundRegionOperationNode ptr	**new_width_check**	(int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)	Creates a node providing a width check.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[static]	new CompoundRegionOpera ptr	**new_minkowsky_sum**	(CompoundRegionOper ptr input, const Edge e)	Use of this method is deprecated. Use new_minkowski_sum instead
[static]	new CompoundRegionOperationNode ptr	**new_minkowsky_sum**	(CompoundRegionOperationNode ptr input, const Polygon p)	Use of this method is deprecated. Use new_minkowski_sum instead
[static]	new CompoundRegionOpera ptr	**new_minkowsky_sum**	(CompoundRegionOper ptr input, const Box p)	Use of this method is deprecated. Use new_minkowski_sum instead
[static]	new CompoundRegionOperationNode ptr	**new_minkowsky_sum**	(CompoundRegionOperationNode ptr input, Point[] p)	Use of this method is deprecated. Use new_minkowski_sum instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

description

Signature: *[const]* string **description**

Description: Gets the description for this node

Python specific notes:
The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string d)

Description: Sets the description for this node

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

distance

Signature: *[const]* int **distance**

Description: Gets the distance value for this node

Python specific notes:
The object exposes a readable attribute 'distance'. This is the getter.

distance=

Signature: void **distance=** (int d)

Description: Sets the distance value for this nodeUsually it's not required to provide a distance because the nodes compute a distance based on their operation. If necessary you can supply a distance. The processor will use this distance or the computed one, whichever is larger.

Python specific notes:
The object exposes a writable attribute 'distance'. This is the setter.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new CompoundRegionOperationNode ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

new_area_filter

Signature: *[static]* new CompoundRegionOperationNode ptr **new_area_filter** (CompoundRegionOperationNode ptr input, bool inverse = false, long amin = 0, long amax = max)

Description: Creates a node filtering the input by area.

This node renders the input if the area is between amin and amax (exclusively). If 'inverse' is set to true, the input shape is returned if the area is less than amin (exclusively) or larger than amax (inclusively).

new_area_sum_filter

Signature: *[static]* new CompoundRegionOperationNode ptr **new_area_sum_filter** (CompoundRegionOperationNode ptr input, bool inverse = false, long amin = 0, long amax = max)

	Description: Creates a node filtering the input by area sum. Like new_area_filter, but applies to the sum of all shapes in the current set.
new_bbox_filter	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_bbox_filter** (CompoundRegionOperationNode ptr input, CompoundRegionOperationNode::ParameterType parameter, bool inverse = false, unsigned int pmin = 0, unsigned int pmax = max) **Description**: Creates a node filtering the input by bounding box parameters. This node renders the input if the specified bounding box parameter of the input shape is between pmin and pmax (exclusively). If 'inverse' is set to true, the input shape is returned if the parameter is less than pmin (exclusively) or larger than pmax (inclusively).
new_case	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_case** (CompoundRegionOperationNode ptr[] inputs) **Description**: Creates a 'switch ladder' (case statement) compound operation node. The inputs are treated as a sequence of condition/result pairs: c1,r1,c2,r2 etc. If there is an odd number of inputs, the last element is taken as the default result. The implementation will evaluate c1 and if not empty, will render r1. Otherwise, c2 will be evaluated and r2 rendered if c2 isn't empty etc. If none of the conditions renders a non-empty set and a default result is present, the default will be returned. Otherwise, the result is empty.
new_centers	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_centers** (CompoundRegionOperationNode ptr input, unsigned int length, double fraction) **Description**: Creates a node delivering a part at the center of each input edge.
new_convex_decomposition	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_convex_decomposition** (CompoundRegionOperationNode ptr input, PreferredOrientation mode) **Description**: Creates a node providing a composition into convex pieces.
new_corners_as_dots	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_corners_as_dots** (CompoundRegionOperationNode ptr input, double angle_min, bool include_angle_min, double angle_max, bool include_angle_max) **Description**: Creates a node turning corners into dots (single-point edges).
new_corners_as_edge_pairs	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_corners_as_edge_pairs** (CompoundRegionOperationNode ptr input, double angle_min, bool include_angle_min, double angle_max, bool include_angle_max) **Description**: Creates a node turning corners into edge pairs containing the two edges adjacent to the corner. The first edge will be the incoming edge and the second one the outgoing edge. This feature has been introduced in version 0.27.1.
new_corners_as_rectangles	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_corners_as_rectangles** (CompoundRegionOperationNode ptr input, double angle_min, bool include_angle_min, double angle_max, bool include_angle_max, int dim) **Description**: Creates a node turning corners into rectangles.
new_count_filter	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_count_filter** (CompoundRegionOperationNode ptr inputs, bool invert = false, unsigned long min_count = 0, unsigned long max_count = 18446744073709551615)

	Description: Creates a node selecting results but their shape count.
new_edge_length_filter	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_edge_length_filter** (CompoundRegionOperationNode ptr input, bool inverse = false, unsigned int lmin = 0, unsigned int lmax = max) **Description**: Creates a node filtering edges by their length.
new_edge_length_sum_filter	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_edge_length_sum_filter** (CompoundRegionOperationNode ptr input, bool inverse = false, unsigned int lmin = 0, unsigned int lmax = max) **Description**: Creates a node filtering edges by their length sum (over the local set).
new_edge_orientation_filter	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_edge_orientation_filter** (CompoundRegionOperationNode ptr input, bool inverse, double amin, bool include_amin, double amax, bool include_amax) **Description**: Creates a node filtering edges by their orientation.
new_edge_pair_to_first_edges	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_edge_pair_to_first_edges** (CompoundRegionOperationNode ptr input) **Description**: Creates a node delivering the first edge of each edges pair.
new_edge_pair_to_second_edges	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_edge_pair_to_second_edges** (CompoundRegionOperationNode ptr input) **Description**: Creates a node delivering the second edge of each edges pair.
new_edges	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_edges** (CompoundRegionOperationNode ptr input, EdgeMode mode = All) **Description**: Creates a node converting polygons into its edges. The 'mode' argument allows selecting specific edges when generating edges from a polygon. See EdgeMode for the various options. By default, all edges are generated from polygons. The 'mode' argument has been added in version 0.29.
new_empty	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_empty** (CompoundRegionOperationNode::ResultType type) **Description**: Creates a node delivering an empty result of the given type
new_enclosed_check	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_enclosed_check** (CompoundRegionOperationNode ptr other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false) **Description**: Creates a node providing an enclosed (secondary enclosing primary) check. This method has been added in version 0.27.5. The zero_distance_mode argument has been inserted in version 0.29.
new_enclosing	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_enclosing** (CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b, bool inverse = false, unsigned long min_count = 1, unsigned long max_count = unlimited)

	Description: Creates a node representing an inside selection operation between the inputs.
new_enclosing_check	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_enclosing_check** (CompoundRegionOperationNode ptr other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false) **Description**: Creates a node providing an inside (enclosure) check. The zero_distance_mode argument has been inserted in version 0.29.
new_end_segments	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_end_segments** (CompoundRegionOperationNode ptr input, unsigned int length, double fraction) **Description**: Creates a node delivering a part at the end of each input edge.
new_extended	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_extended** (CompoundRegionOperationNode ptr input, int ext_b, int ext_e, int ext_o, int ext_i) **Description**: Creates a node delivering a polygonized version of the edges with the four extension parameters.
new_extended_in	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_extended_in** (CompoundRegionOperationNode ptr input, int e) **Description**: Creates a node delivering a polygonized, inside-extended version of the edges.
new_extended_out	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_extended_out** (CompoundRegionOperationNode ptr input, int e) **Description**: Creates a node delivering a polygonized, inside-extended version of the edges.
new_extents	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_extents** (CompoundRegionOperationNode ptr input, int e = 0) **Description**: Creates a node returning the extents of the objects. The 'e' parameter provides a generic enlargement which is applied to the boxes. This is helpful to cover dot-like edges or edge pairs in the input.
new_foreign	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_foreign** **Description**: Creates a node object representing the primary input without the current polygon
new_geometrical_boolean	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_geometrical_boolean** (CompoundRegionOperationNode::GeometricalOp op, CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b) **Description**: Creates a node representing a geometrical boolean operation between the inputs.
new_hole_count_filter	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_hole_count_filter** (CompoundRegionOperationNode ptr input, bool inverse = false, unsigned long hmin = 0, unsigned long hmax = max) **Description**: Creates a node filtering the input by number of holes per polygon.

	This node renders the input if the hole count is between hmin and hmax (exclusively). If 'inverse' is set to true, the input shape is returned if the hole count is less than hmin (exclusively) or larger than hmax (inclusively).
new_holes	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_holes** (CompoundRegionOperationNode ptr input) **Description**: Creates a node extracting the holes from polygons.
new_hulls	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_hulls** (CompoundRegionOperationNode ptr input) **Description**: Creates a node extracting the hulls from polygons.
new_inside	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_inside** (CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b, bool inverse = false) **Description**: Creates a node representing an inside selection operation between the inputs.
new_interacting	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_interacting** (CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b, bool inverse = false, unsigned long min_count = 1, unsigned long max_count = unlimited) **Description**: Creates a node representing an interacting selection operation between the inputs.
new_isolated_check	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_isolated_check** (int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false) **Description**: Creates a node providing a isolated polygons (space between different polygons) check. The zero_distance_mode argument has been inserted in version 0.29.
new_join	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_join** (CompoundRegionOperationNode ptr[] inputs) **Description**: Creates a node that joins the inputs.
new_logical_boolean	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_logical_boolean** (CompoundRegionOperationNode::LogicalOp op, bool invert, CompoundRegionOperationNode ptr[] inputs) **Description**: Creates a node representing a logical boolean operation between the inputs. A logical AND operation will evaluate the arguments and render the subject shape when all arguments are non-empty. The logical OR operation will evaluate the arguments and render the subject shape when one argument is non-empty. Setting 'inverse' to true will reverse the result and return the subject shape when one argument is empty in the AND case and when all arguments are empty in the OR case.
new_merged	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_merged** (CompoundRegionOperationNode ptr input, bool min_coherence = false, unsigned int min_wc = 0)

	Description: Creates a node providing merged input polygons.
new_minkowski_sum	**(1) Signature**: *[static]* new CompoundRegionOperationNode ptr **new_minkowski_sum** (CompoundRegionOperationNode ptr input, const Edge e) **Description**: Creates a node providing a Minkowski sum with an edge. **(2) Signature**: *[static]* new CompoundRegionOperationNode ptr **new_minkowski_sum** (CompoundRegionOperationNode ptr input, const Polygon p) **Description**: Creates a node providing a Minkowski sum with a polygon. **(3) Signature**: *[static]* new CompoundRegionOperationNode ptr **new_minkowski_sum** (CompoundRegionOperationNode ptr input, const Box p) **Description**: Creates a node providing a Minkowski sum with a box. **(4) Signature**: *[static]* new CompoundRegionOperationNode ptr **new_minkowski_sum** (CompoundRegionOperationNode ptr input, Point[] p) **Description**: Creates a node providing a Minkowski sum with a point sequence forming a contour.
new_minkowsky_sum	**(1) Signature**: *[static]* new CompoundRegionOperationNode ptr **new_minkowsky_sum** (CompoundRegionOperationNode ptr input, const Edge e) **Description**: Creates a node providing a Minkowski sum with an edge. Use of this method is deprecated. Use new_minkowski_sum instead **(2) Signature**: *[static]* new CompoundRegionOperationNode ptr **new_minkowsky_sum** (CompoundRegionOperationNode ptr input, const Polygon p) **Description**: Creates a node providing a Minkowski sum with a polygon. Use of this method is deprecated. Use new_minkowski_sum instead **(3) Signature**: *[static]* new CompoundRegionOperationNode ptr **new_minkowsky_sum** (CompoundRegionOperationNode ptr input, const Box p) **Description**: Creates a node providing a Minkowski sum with a box. Use of this method is deprecated. Use new_minkowski_sum instead **(4) Signature**: *[static]* new CompoundRegionOperationNode ptr **new_minkowsky_sum** (CompoundRegionOperationNode ptr input, Point[] p) **Description**: Creates a node providing a Minkowski sum with a point sequence forming a contour. Use of this method is deprecated. Use new_minkowski_sum instead
new_notch_check	**Signature**: *[static]* new CompoundRegionOperationNode ptr **new_notch_check** (int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false) **Description**: Creates a node providing a intra-polygon space check. The zero_distance_mode argument has been inserted in version 0.29.

new_outside

Signature: *[static]* new CompoundRegionOperationNode ptr **new_outside** (CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b, bool inverse = false)

Description: Creates a node representing an outside selection operation between the inputs.

new_overlap_check

Signature: *[static]* new CompoundRegionOperationNode ptr **new_overlap_check** (CompoundRegionOperationNode ptr other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)

Description: Creates a node providing an overlap check.

The zero_distance_mode argument has been inserted in version 0.29.

new_overlapping

Signature: *[static]* new CompoundRegionOperationNode ptr **new_overlapping** (CompoundRegionOperationNode ptr a, CompoundRegionOperationNode ptr b, bool inverse = false, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Creates a node representing an overlapping selection operation between the inputs.

new_perimeter_filter

Signature: *[static]* new CompoundRegionOperationNode ptr **new_perimeter_filter** (CompoundRegionOperationNode ptr input, bool inverse = false, unsigned long pmin = 0, unsigned long pmax = max)

Description: Creates a node filtering the input by perimeter.

This node renders the input if the perimeter is between pmin and pmax (exclusively). If 'inverse' is set to true, the input shape is returned if the perimeter is less than pmin (exclusively) or larger than pmax (inclusively).

new_perimeter_sum_filter

Signature: *[static]* new CompoundRegionOperationNode ptr **new_perimeter_sum_filter** (CompoundRegionOperationNode ptr input, bool inverse = false, unsigned long amin = 0, unsigned long amax = max)

Description: Creates a node filtering the input by area sum.

Like new_perimeter_filter, but applies to the sum of all shapes in the current set.

new_polygon_breaker

Signature: *[static]* new CompoundRegionOperationNode ptr **new_polygon_breaker** (CompoundRegionOperationNode ptr input, unsigned long max_vertex_count, double max_area_ratio)

Description: Creates a node providing a composition into parts with less than the given number of points and a smaller area ratio.

new_polygons

Signature: *[static]* new CompoundRegionOperationNode ptr **new_polygons** (CompoundRegionOperationNode ptr input, int e = 0)

Description: Creates a node converting the input to polygons.

e: The enlargement parameter when converting edges or edge pairs to polygons.

new_primary

Signature: *[static]* new CompoundRegionOperationNode ptr **new_primary**

Description: Creates a node object representing the primary input

new_ratio_filter

Signature: *[static]* new CompoundRegionOperationNode ptr **new_ratio_filter** (CompoundRegionOperationNode ptr input, CompoundRegionOperationNode::RatioParameterType parameter, bool inverse = false, double pmin = 0, bool pmin_included = true, double pmax = max, bool pmax_included = true)

Description: Creates a node filtering the input by ratio parameters.

This node renders the input if the specified ratio parameter of the input shape is between pmin and pmax. If 'pmin_included' is true, the range will include pmin. Same for 'pmax_included' and pmax. If 'inverse' is set to true, the input shape is returned if the parameter is not within the specified range.

new_rectangle_filter

Signature: *[static]* new CompoundRegionOperationNode ptr **new_rectangle_filter** (CompoundRegionOperationNode ptr input, bool is_square = false, bool inverse = false)

Description: Creates a node filtering the input for rectangular or square shapes.

If 'is_square' is true, only squares will be selected. If 'inverse' is true, the non-rectangle/non-square shapes are returned.

new_rectilinear_filter

Signature: *[static]* new CompoundRegionOperationNode ptr **new_rectilinear_filter** (CompoundRegionOperationNode ptr input, bool inverse = false)

Description: Creates a node filtering the input for rectilinear shapes (or non-rectilinear ones with 'inverse' set to 'true').

new_relative_extents

Signature: *[static]* new CompoundRegionOperationNode ptr **new_relative_extents** (CompoundRegionOperationNode ptr input, double fx1, double fy1, double fx2, double fy2, int dx, int dy)

Description: Creates a node returning markers at specified locations of the extent (e.g. at the center).

new_relative_extents_as_edge

Signature: *[static]* new CompoundRegionOperationNode ptr **new_relative_extents_as_edges** (CompoundRegionOperationNode ptr input, double fx1, double fy1, double fx2, double fy2)

Description: Creates a node returning edges at specified locations of the extent (e.g. at the center).

new_rounded_corners

Signature: *[static]* new CompoundRegionOperationNode ptr **new_rounded_corners** (CompoundRegionOperationNode ptr input, double rinner, double router, unsigned int n)

Description: Creates a node generating rounded corners.

rinner: The inner corner radius.@param router The outer corner radius.@param n The number if points per full circle.

new_secondary

Signature: *[static]* new CompoundRegionOperationNode ptr **new_secondary** (Region ptr region)

Description: Creates a node object representing the secondary input from the given region

new_separation_check

Signature: *[static]* new CompoundRegionOperationNode ptr **new_separation_check** (CompoundRegionOperationNode ptr other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)

Description: Creates a node providing a separation check.

The zero_distance_mode argument has been inserted in version 0.29.

new_sized

Signature: *[static]* new CompoundRegionOperationNode ptr **new_sized** (CompoundRegionOperationNode ptr input, int dx, int dy, unsigned int mode)

Description: Creates a node providing sizing.

new_smoothed

Signature: *[static]* new CompoundRegionOperationNode ptr **new_smoothed** (CompoundRegionOperationNode ptr input, int d, bool keep_hv = false)

Description: Creates a node smoothing the polygons.

d: The tolerance to be applied for the smoothing.

keep_hv: If true, horizontal and vertical edges are maintained.

new_space_check

Signature: *[static]* new CompoundRegionOperationNode ptr **new_space_check** (int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)

Description: Creates a node providing a space check.

The zero_distance_mode argument has been inserted in version 0.29.

new_start_segments

Signature: *[static]* new CompoundRegionOperationNode ptr **new_start_segments** (CompoundRegionOperationNode ptr input, unsigned int length, double fraction)

Description: Creates a node delivering a part at the beginning of each input edge.

new_strange_polygons_filter

Signature: *[static]* new CompoundRegionOperationNode ptr **new_strange_polygons_filter** (CompoundRegionOperationNode ptr input)

Description: Creates a node extracting strange polygons.

'strange polygons' are ones which cannot be oriented - e.g. '8' shape polygons.

new_trapezoid_decomposition

Signature: *[static]* new CompoundRegionOperationNode ptr **new_trapezoid_decomposition** (CompoundRegionOperationNode ptr input, TrapezoidDecompositionMode mode)

Description: Creates a node providing a composition into trapezoids.

new_width_check

Signature: *[static]* new CompoundRegionOperationNode ptr **new_width_check** (int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max., bool shielded = true, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching, bool negative = false)

Description: Creates a node providing a width check.

The zero_distance_mode argument has been inserted in version 0.29.

result_type

Signature: *[const]* CompoundRegionOperationNode::ResultType **result_type**

Description: Gets the result type of this node

4.26. API reference - Class CompoundRegionOperationNode::LogicalOp

Notation used in Ruby API documentation

Module: db

Description: This class represents the CompoundRegionOperationNode::LogicalOp enum

This class is equivalent to the class CompoundRegionOperationNode::LogicalOp

This enum has been introduced in version 0.27.

Public constructors

new CompoundRegionOperationNode::LogicalOp ptr	**new**	(int i)	Creates an enum from an integer value
new CompoundRegionOperationNode::LogicalOp ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const CompoundRegionOperatic other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const CompoundRegionOperatic other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const CompoundRegionOperatic other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	CompoundRegionOperationNode::Logical(	**LogAnd**	Indicates a logical '&&' (and).
[static,const]	CompoundRegionOperationNode::LogicalOp	**LogOr**	Indicates a logical '||' (or).

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const CompoundRegionOperationNode::LogicalOp other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const CompoundRegionOperationNode::LogicalOp other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const CompoundRegionOperationNode::LogicalOp other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
LogAnd	**Signature**: *[static,const]* CompoundRegionOperationNode::LogicalOp **LogAnd** **Description**: Indicates a logical '&&' (and). **Python specific notes:** The object exposes a readable attribute 'LogAnd'. This is the getter.
LogOr	**Signature**: *[static,const]* CompoundRegionOperationNode::LogicalOp **LogOr** **Description**: Indicates a logical '||' (or). **Python specific notes:** The object exposes a readable attribute 'LogOr'. This is the getter.
hash	**Signature**: *[const]* int **hash** **Description**: Gets the hash value from the enum **Python specific notes:** This method is also available as 'hash(object)'.
inspect	**Signature**: *[const]* string **inspect** **Description**: Converts an enum to a visual string **Python specific notes:** This method is also available as 'repr(object)'.
new	**(1) Signature**: *[static]* new CompoundRegionOperationNode::LogicalOp ptr **new** (int i) **Description**: Creates an enum from an integer value **Python specific notes:** This method is the default initializer of the object.

(2) Signature: *[static]* new CompoundRegionOperationNode::LogicalOp ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.27. API reference - Class CompoundRegionOperationNode::GeometricalOp

Notation used in Ruby API documentation

Module: db

Description: This class represents the CompoundRegionOperationNode::GeometricalOp enum

This class is equivalent to the class CompoundRegionOperationNode::GeometricalOp

This enum has been introduced in version 0.27.

Public constructors

new CompoundRegionOperationNode::GeometricalOp ptr	**new**	(int i)	Creates an enum from an integer value
new CompoundRegionOperationNode::GeometricalOp ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const CompoundRegionOperatic other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const CompoundRegionOperatic other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const CompoundRegionOperatic other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	CompoundRegionOperationNode::Geometric	**And**	Indicates a geometrical '&' (and).
[static,const]	CompoundRegionOperationNode::GeometricalOp	**Not**	Indicates a geometrical '-' (not).

[static,const]	CompoundRegionOperationNode::Geometric **Or**	Indicates a geometrical '|' (or).
[static,const]	CompoundRegionOperationNode::GeometricalOp **Xor**	Indicates a geometrical '^' (xor).

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const CompoundRegionOperationNode::GeometricalOp other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const CompoundRegionOperationNode::GeometricalOp other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const CompoundRegionOperationNode::GeometricalOp other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
And	**Signature**: *[static,const]* CompoundRegionOperationNode::GeometricalOp **And** **Description**: Indicates a geometrical '&' (and). **Python specific notes:** The object exposes a readable attribute 'And'. This is the getter.
Not	**Signature**: *[static,const]* CompoundRegionOperationNode::GeometricalOp **Not** **Description**: Indicates a geometrical '-' (not). **Python specific notes:** The object exposes a readable attribute 'Not'. This is the getter.
Or	**Signature**: *[static,const]* CompoundRegionOperationNode::GeometricalOp **Or** **Description**: Indicates a geometrical '|' (or). **Python specific notes:** The object exposes a readable attribute 'Or'. This is the getter.
Xor	**Signature**: *[static,const]* CompoundRegionOperationNode::GeometricalOp **Xor** **Description**: Indicates a geometrical '^' (xor). **Python specific notes:** The object exposes a readable attribute 'Xor'. This is the getter.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

new

(1) Signature: *[static]* new CompoundRegionOperationNode::GeometricalOp ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new CompoundRegionOperationNode::GeometricalOp ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.28. API reference - Class CompoundRegionOperationNode::ResultType

Notation used in Ruby API documentation

Module: db

Description: This class represents the CompoundRegionOperationNode::ResultType enum

This class is equivalent to the class CompoundRegionOperationNode::ResultType

This enum has been introduced in version 0.27.

Public constructors

new CompoundRegionOperationNode::ResultType ptr	**new**	(int i)	Creates an enum from an integer value
new CompoundRegionOperationNode::ResultType ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const CompoundRegionOperatic other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const CompoundRegionOperatic other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const CompoundRegionOperatic other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	CompoundRegionOperationNode::ResultT	**EdgePairs**	Indicates edge pair result type.
[static,const]	CompoundRegionOperationNode::ResultType	**Edges**	Indicates edge result type.

[static,const]	CompoundRegionOperationNode::ResultT	**Region**	Indicates polygon result type.

Detailed description

!=

(1) Signature: *[const]* bool **!=** (const CompoundRegionOperationNode::ResultType other)
Description: Compares two enums for inequality

(2) Signature: *[const]* bool **!=** (int other)
Description: Compares an enum with an integer for inequality

<

(1) Signature: *[const]* bool **<** (const CompoundRegionOperationNode::ResultType other)
Description: Returns true if the first enum is less (in the enum symbol order) than the second

(2) Signature: *[const]* bool **<** (int other)
Description: Returns true if the enum is less (in the enum symbol order) than the integer value

==

(1) Signature: *[const]* bool **==** (const CompoundRegionOperationNode::ResultType other)
Description: Compares two enums

(2) Signature: *[const]* bool **==** (int other)
Description: Compares an enum with an integer value

EdgePairs

Signature: *[static,const]* CompoundRegionOperationNode::ResultType **EdgePairs**
Description: Indicates edge pair result type.
Python specific notes:
The object exposes a readable attribute 'EdgePairs'. This is the getter.

Edges

Signature: *[static,const]* CompoundRegionOperationNode::ResultType **Edges**
Description: Indicates edge result type.
Python specific notes:
The object exposes a readable attribute 'Edges'. This is the getter.

Region

Signature: *[static,const]* CompoundRegionOperationNode::ResultType **Region**
Description: Indicates polygon result type.
Python specific notes:
The object exposes a readable attribute 'Region'. This is the getter.

hash

Signature: *[const]* int **hash**
Description: Gets the hash value from the enum
Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**
Description: Converts an enum to a visual string
Python specific notes:

This method is also available as 'repr(object)'.

new

(1) Signature: *[static]* new CompoundRegionOperationNode::ResultType ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new CompoundRegionOperationNode::ResultType ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.29. API reference - Class CompoundRegionOperationNode::ParameterType

Notation used in Ruby API documentation

Module: db

Description: This class represents the parameter type enum used in \CompoundRegionOperationNode#new_bbox_filter

This class is equivalent to the class CompoundRegionOperationNode::ParameterType

This enum has been introduced in version 0.27.

Public constructors

new CompoundRegionOperationNode::ParameterType ptr	**new**	(int i)	Creates an enum from an integer value
new CompoundRegionOperationNode::ParameterType ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const CompoundRegionOperatic other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const CompoundRegionOperatic other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const CompoundRegionOperatic other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	CompoundRegionOperationNode::Pa	**BoxAverageDim**	Measures the average of width and height of the bounding box

[static,const]	CompoundRegionOperationNode::Parameter	**BoxHeight**	Measures the height of the bounding box
[static,const]	CompoundRegionOperationNode::Pa	**BoxMaxDim**	Measures the maximum dimension of the bounding box
[static,const]	CompoundRegionOperationNode::Parameter	**BoxMinDim**	Measures the minimum dimension of the bounding box
[static,const]	CompoundRegionOperationNode::Pa	**BoxWidth**	Measures the width of the bounding box

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const CompoundRegionOperationNode::ParameterType other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const CompoundRegionOperationNode::ParameterType other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const CompoundRegionOperationNode::ParameterType other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
BoxAverageDim	**Signature**: *[static,const]* CompoundRegionOperationNode::ParameterType **BoxAverageDim** **Description**: Measures the average of width and height of the bounding box **Python specific notes:** The object exposes a readable attribute 'BoxAverageDim'. This is the getter.
BoxHeight	**Signature**: *[static,const]* CompoundRegionOperationNode::ParameterType **BoxHeight** **Description**: Measures the height of the bounding box **Python specific notes:** The object exposes a readable attribute 'BoxHeight'. This is the getter.
BoxMaxDim	**Signature**: *[static,const]* CompoundRegionOperationNode::ParameterType **BoxMaxDim** **Description**: Measures the maximum dimension of the bounding box **Python specific notes:** The object exposes a readable attribute 'BoxMaxDim'. This is the getter.

BoxMinDim

Signature: *[static,const]* CompoundRegionOperationNode::ParameterType **BoxMinDim**

Description: Measures the minimum dimension of the bounding box

Python specific notes:
The object exposes a readable attribute 'BoxMinDim'. This is the getter.

BoxWidth

Signature: *[static,const]* CompoundRegionOperationNode::ParameterType **BoxWidth**

Description: Measures the width of the bounding box

Python specific notes:
The object exposes a readable attribute 'BoxWidth'. This is the getter.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

new

(1) Signature: *[static]* new CompoundRegionOperationNode::ParameterType ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new CompoundRegionOperationNode::ParameterType ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.30. API reference - Class CompoundRegionOperationNode::RatioParameterType

Notation used in Ruby API documentation

Module: db

Description: This class represents the parameter type enum used in \CompoundRegionOperationNode#new_ratio_filter

This class is equivalent to the class CompoundRegionOperationNode::RatioParameterType

This enum has been introduced in version 0.27.

Public constructors

new CompoundRegionOperationNode::RatioParameter ptr	**new**	(int i)	Creates an enum from an integer value
new CompoundRegionOperationNode::RatioParameterType ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const CompoundRegionOperatic other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const CompoundRegionOperatic other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const CompoundRegionOperatic other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	CompoundRegionOperationNode::Ra	**AreaRatio**	Measures the area ratio (bounding box area / polygon area)

[static,const]	CompoundRegionOperationNode::RatioParameterType **AspectRatio**	Measures the aspect ratio of the bounding box (larger / smaller dimension)
[static,const]	CompoundRegionOperationNode::RatioParameterType **RelativeHeight**	Measures the relative height (height / width)

Detailed description

!=

(1) Signature: *[const]* bool **!=** (const CompoundRegionOperationNode::RatioParameterType other)
Description: Compares two enums for inequality

(2) Signature: *[const]* bool **!=** (int other)
Description: Compares an enum with an integer for inequality

<

(1) Signature: *[const]* bool **<** (const CompoundRegionOperationNode::RatioParameterType other)
Description: Returns true if the first enum is less (in the enum symbol order) than the second

(2) Signature: *[const]* bool **<** (int other)
Description: Returns true if the enum is less (in the enum symbol order) than the integer value

==

(1) Signature: *[const]* bool **==** (const CompoundRegionOperationNode::RatioParameterType other)
Description: Compares two enums

(2) Signature: *[const]* bool **==** (int other)
Description: Compares an enum with an integer value

AreaRatio

Signature: *[static,const]* CompoundRegionOperationNode::RatioParameterType **AreaRatio**
Description: Measures the area ratio (bounding box area / polygon area)
Python specific notes:
The object exposes a readable attribute 'AreaRatio'. This is the getter.

AspectRatio

Signature: *[static,const]* CompoundRegionOperationNode::RatioParameterType **AspectRatio**
Description: Measures the aspect ratio of the bounding box (larger / smaller dimension)
Python specific notes:
The object exposes a readable attribute 'AspectRatio'. This is the getter.

RelativeHeight

Signature: *[static,const]* CompoundRegionOperationNode::RatioParameterType **RelativeHeight**
Description: Measures the relative height (height / width)
Python specific notes:
The object exposes a readable attribute 'RelativeHeight'. This is the getter.

hash

Signature: *[const]* int **hash**
Description: Gets the hash value from the enum
Python specific notes:
This method is also available as 'hash(object)'.

inspect	**Signature**: *[const]* string **inspect** **Description**: Converts an enum to a visual string **Python specific notes:** This method is also available as 'repr(object)'.
new	**(1) Signature**: *[static]* new CompoundRegionOperationNode::RatioParameterType ptr **new** (int i) **Description**: Creates an enum from an integer value **Python specific notes:** This method is the default initializer of the object. **(2) Signature**: *[static]* new CompoundRegionOperationNode::RatioParameterType ptr **new** (string s) **Description**: Creates an enum from a string value **Python specific notes:** This method is the default initializer of the object.
to_i	**Signature**: *[const]* int **to_i** **Description**: Gets the integer value from the enum **Python specific notes:** This method is also available as 'int(object)'.
to_s	**Signature**: *[const]* string **to_s** **Description**: Gets the symbolic string from an enum **Python specific notes:** This method is also available as 'str(object)'.

4.31. API reference - Class TrapezoidDecompositionMode

Notation used in Ruby API documentation

Module: db

Description: This class represents the TrapezoidDecompositionMode enum used within trapezoid decomposition

This enum has been introduced in version 0.27.

Public constructors

new TrapezoidDecompositionMode ptr	**new**	(int i)	Creates an enum from an integer value
new TrapezoidDecompositionMode ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const TrapezoidDecompo other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const TrapezoidDecompo other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const TrapezoidDecompo other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const TrapezoidDecompo other)	Assigns another object to self

[const]	new TrapezoidDecompositionMode ptr	**dup**	Creates a copy of self
[const]	int	**hash**	Gets the hash value from the enum
[const]	string	**inspect**	Converts an enum to a visual string
[const]	int	**to_i**	Gets the integer value from the enum
[const]	string	**to_s**	Gets the symbolic string from an enum

Public static methods and constants

[static,const]	TrapezoidDecompositionMode	**TD_htrapezoids**	Indicates horizontal trapezoid decomposition.
[static,const]	TrapezoidDecompositionMode	**TD_simple**	Indicates unspecific decomposition.
[static,const]	TrapezoidDecompositionMode	**TD_vtrapezoids**	Indicates vertical trapezoid decomposition.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const TrapezoidDecompositionMode other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const TrapezoidDecompositionMode other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const TrapezoidDecompositionMode other) **Description**: Compares two enums

(2) Signature: *[const]* bool **==** (int other)

Description: Compares an enum with an integer value

TD_htrapezoids

Signature: *[static,const]* TrapezoidDecompositionMode **TD_htrapezoids**

Description: Indicates horizontal trapezoid decomposition.

Python specific notes:
The object exposes a readable attribute 'TD_htrapezoids'. This is the getter.

TD_simple

Signature: *[static,const]* TrapezoidDecompositionMode **TD_simple**

Description: Indicates unspecific decomposition.

Python specific notes:
The object exposes a readable attribute 'TD_simple'. This is the getter.

TD_vtrapezoids

Signature: *[static,const]* TrapezoidDecompositionMode **TD_vtrapezoids**

Description: Indicates vertical trapezoid decomposition.

Python specific notes:
The object exposes a readable attribute 'TD_vtrapezoids'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const TrapezoidDecompositionMode other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new TrapezoidDecompositionMode ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:

	This method is also available as 'repr(object)'.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**(1) Signature**: *[static]* new TrapezoidDecompositionMode ptr **new** (int i) **Description**: Creates an enum from an integer value **Python specific notes:** This method is the default initializer of the object. **(2) Signature**: *[static]* new TrapezoidDecompositionMode ptr **new** (string s) **Description**: Creates an enum from a string value **Python specific notes:** This method is the default initializer of the object.
to_i	**Signature**: *[const]* int **to_i** **Description**: Gets the integer value from the enum **Python specific notes:** This method is also available as 'int(object)'.
to_s	**Signature**: *[const]* string **to_s** **Description**: Gets the symbolic string from an enum **Python specific notes:** This method is also available as 'str(object)'.

4.32. API reference - Class PreferredOrientation

Notation used in Ruby API documentation

Module: db

Description: This class represents the PreferredOrientation enum used within polygon decomposition

This enum has been introduced in version 0.27.

Public constructors

new PreferredOrientation ptr	**new**	(int i)	Creates an enum from an integer value
new PreferredOrientation ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const PreferredOrientati other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const PreferredOrientati other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const PreferredOrientati other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const PreferredOrientati other)	Assigns another object to self

[const]	new PreferredOrientation ptr	**dup**	Creates a copy of self
[const]	int	**hash**	Gets the hash value from the enum
[const]	string	**inspect**	Converts an enum to a visual string
[const]	int	**to_i**	Gets the integer value from the enum
[const]	string	**to_s**	Gets the symbolic string from an enum

Public static methods and constants

[static,const]	PreferredOrientation	**PO_any**	Indicates any orientation.
[static,const]	PreferredOrientation	**PO_horizontal**	Indicates horizontal orientation.
[static,const]	PreferredOrientation	**PO_htrapezoids**	Indicates horizontal trapezoid decomposition.
[static,const]	PreferredOrientation	**PO_vertical**	Indicates vertical orientation.
[static,const]	PreferredOrientation	**PO_vtrapezoids**	Indicates vertical trapezoid decomposition.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const PreferredOrientation other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const PreferredOrientation other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value

==	**(1) Signature**: *[const]* bool **==** (const PreferredOrientation other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
PO_any	**Signature**: *[static,const]* PreferredOrientation **PO_any** **Description**: Indicates any orientation. **Python specific notes:** The object exposes a readable attribute 'PO_any'. This is the getter.
PO_horizontal	**Signature**: *[static,const]* PreferredOrientation **PO_horizontal** **Description**: Indicates horizontal orientation. **Python specific notes:** The object exposes a readable attribute 'PO_horizontal'. This is the getter.
PO_htrapezoids	**Signature**: *[static,const]* PreferredOrientation **PO_htrapezoids** **Description**: Indicates horizontal trapezoid decomposition. **Python specific notes:** The object exposes a readable attribute 'PO_htrapezoids'. This is the getter.
PO_vertical	**Signature**: *[static,const]* PreferredOrientation **PO_vertical** **Description**: Indicates vertical orientation. **Python specific notes:** The object exposes a readable attribute 'PO_vertical'. This is the getter.
PO_vtrapezoids	**Signature**: *[static,const]* PreferredOrientation **PO_vtrapezoids** **Description**: Indicates vertical trapezoid decomposition. **Python specific notes:** The object exposes a readable attribute 'PO_vtrapezoids'. This is the getter.
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const PreferredOrientation other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new PreferredOrientation ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

(1) Signature: *[static]* new PreferredOrientation ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new PreferredOrientation ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.33. API reference - Class Edge

Notation used in Ruby API documentation

Module: db

Description: An edge class

An edge is a connection between points, usually participating in a larger context such as a polygon. An edge has a defined direction (from p1 to p2). Edges play a role in the database as parts of polygons and to describe a line through both points. Although supported, edges are rarely used as individual database objects.

See The Database API for more details about the database objects like the Edge class.

Public constructors

new Edge ptr	**new**	(const DEdge dedge)	Creates an integer coordinate edge from a floating-point coordinate edge
new Edge ptr	**new**		Default constructor: creates a degenerated edge 0,0 to 0,0
new Edge ptr	**new**	(int x1, int y1, int x2, int y2)	Constructor with two coordinates given as single values
new Edge ptr	**new**	(const Point p1, const Point p2)	Constructor with two points

Public methods

[const]	bool	**!=**	(const Edge e)	Inequality test
[const]	Edge	*****	(double scale_factor)	Scale edge
[const]	bool	**<**	(const Edge e)	Less operator
[const]	bool	**==**	(const Edge e)	Equality test
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Edge other)	Assigns another object to self

[const]	Box	**bbox**		Return the bounding box of the edge.
[const]	variant	**clipped**	(const Box box)	Returns the edge clipped at the given box
[const]	variant	**clipped_line**	(const Box box)	Returns the line through the edge clipped at the given box
[const]	bool	**coincident?**	(const Edge e)	Coincidence check.
[const]	bool	**contains?**	(const Point p)	Tests whether a point is on an edge.
[const]	bool	**contains_excl?**	(const Point p)	Tests whether a point is on an edge excluding the endpoints.
[const]	bool	**crossed_by?**	(const Edge e)	Checks, if the line given by self is crossed by the edge e
[const]	Point	**crossing_point**	(const Edge e)	Returns the crossing point on two edges.
[const]	variant	**cut_point**	(const Edge e)	Returns the intersection point of the lines through the two edges.
[const]	Vector	**d**		Gets the edge extension as a vector.
[const]	int	**distance**	(const Point p)	Gets the distance of the point from the line through the edge.
[const]	unsigned int	**distance_abs**	(const Point p)	Absolute distance between the edge and a point.
[const]	new Edge ptr	**dup**		Creates a copy of self
[const]	int	**dx**		The horizontal extend of the edge.
[const]	unsigned int	**dx_abs**		The absolute value of the horizontal extend of the edge.
[const]	int	**dy**		The vertical extend of the edge.
[const]	unsigned int	**dy_abs**		The absolute value of the vertical extend of the edge.
	Edge	**enlarge**	(const Vector p)	Enlarges the edge.
[const]	Edge	**enlarged**	(const Vector p)	Returns the enlarged edge (does not modify self)
	unsigned int	**euclidian_distance**	(const Point p)	Gets the distance of the point from the the edge.
	Edge	**extend**	(int d)	Extends the edge (modifies self)
[const]	Edge	**extended**	(int d)	Returns the extended edge (does not modify self)
[const]	unsigned long	**hash**		Computes a hash value
[const]	variant	**intersection_point**	(const Edge e)	Returns the intersection point of two edges.

[const]	bool	**intersects?**	(const Edge e)	Intersection test.
[const]	bool	**is_degenerate?**		Test for degenerated edge
[const]	bool	**is_parallel?**	(const Edge e)	Test for being parallel
[const]	unsigned int	**length**		The length of the edge
	Edge	**move**	(const Vector p)	Moves the edge.
	Edge	**move**	(int dx, int dy)	Moves the edge.
[const]	Edge	**moved**	(const Vector p)	Returns the moved edge (does not modify self)
[const]	Edge	**moved**	(int dx, int dy)	Returns the moved edge (does not modify self)
[const]	unsigned int	**ortho_length**		The orthogonal length of the edge ("manhattan-length")
[const]	Point	**p1**		The first point.
	void	**p1=**	(const Point point)	Sets the first point.
[const]	Point	**p2**		The second point.
	void	**p2=**	(const Point point)	Sets the second point.
	Edge	**shift**	(int d)	Shifts the edge (modifies self)
[const]	Edge	**shifted**	(int d)	Returns the shifted edge (does not modify self)
[const]	int	**side_of**	(const Point p)	Indicates at which side the point is located relative to the edge.
[const]	long	**sq_length**		The square of the length of the edge
	Edge	**swap_points**		Swap the points of the edge
[const]	Edge	**swapped_points**		Returns an edge in which both points are swapped
[const]	DEdge	**to_dtype**	(double dbu = 1)	Converts the edge to a floating-point coordinate edge
[const]	string	**to_s**	(double dbu = 0)	Returns a string representing the edge
[const]	Edge	**transformed**	(const ICplxTrans t)	Transform the edge.
[const]	Edge	**transformed**	(const Trans t)	Transform the edge.
[const]	DEdge	**transformed**	(const CplxTrans t)	Transform the edge.

[const]	int	**x1**		Shortcut for p1.x
	void	**x1=**	(int coord)	Sets p1.x
[const]	int	**x2**		Shortcut for p2.x
	void	**x2=**	(int coord)	Sets p2.x
[const]	int	**y1**		Shortcut for p1.y
	void	**y1=**	(int coord)	Sets p1.y
[const]	int	**y2**		Shortcut for p2.y
	void	**y2=**	(int coord)	Sets p2.y

Public static methods and constants

new Edge ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new Edge ptr	**from_dedge**	(const DEdge dedge)	Use of this method is deprecated. Use new instead
[const]	bool	**intersect?**	(const Edge e)	Use of this method is deprecated. Use intersects? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[static]	new Edge ptr	**new_pp**	(const Point p1, const Point p2)	Use of this method is deprecated. Use new instead
[static]	new Edge ptr	**new_xyxy**	(int x1, int y1, int x2, int y2)	Use of this method is deprecated. Use new instead
[const]	DEdge	**transformed_cplx**	(const CplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

!=

Signature: *[const]* bool **!=** (const Edge e)

Description: Inequality test

e: The object to compare against

*****

Signature: *[const]* Edge ***** (double scale_factor)

Description: Scale edge

scale_factor: The scaling factor

Returns: The scaled edge

The * operator scales self with the given factor.

This method has been introduced in version 0.22.

Python specific notes:
This method also implements '__rmul__'.

<

Signature: *[const]* bool **<** (const Edge e)

Description: Less operator

e: The object to compare against

Returns: True, if the edge is 'less' as the other edge with respect to first and second point

==

Signature: *[const]* bool **==** (const Edge e)

Description: Equality test

e: The object to compare against

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Edge other)

Description: Assigns another object to self

bbox

Signature: *[const]* Box **bbox**

Description: Return the bounding box of the edge.

clipped

Signature: *[const]* variant **clipped** (const Box box)

Description: Returns the edge clipped at the given box

box:	The clip box.
Returns:	The clipped edge or nil if the edge does not intersect with the box.

This method has been introduced in version 0.26.2.

clipped_line

Signature: *[const]* variant **clipped_line** (const Box box)

Description: Returns the line through the edge clipped at the given box

box:	The clip box.
Returns:	The part of the line through the box or nil if the line does not intersect with the box.

In contrast to clipped, this method will consider the edge extended infinitely (a "line"). The returned edge will be the part of this line going through the box.

This method has been introduced in version 0.26.2.

coincident?

Signature: *[const]* bool **coincident?** (const Edge e)

Description: Coincidence check.

e:	the edge to test with
Returns:	True if the edges are coincident.

Checks whether a edge is coincident with another edge. Coincidence is defined by being parallel and that at least one point of one edge is on the other edge.

contains?

Signature: *[const]* bool **contains?** (const Point p)

Description: Tests whether a point is on an edge.

p: The point to test with the edge.

Returns: True if the point is on the edge.

A point is on a edge if it is on (or at least closer than a grid point to) the edge.

contains_excl?

Signature: *[const]* bool **contains_excl?** (const Point p)

Description: Tests whether a point is on an edge excluding the endpoints.

p: The point to test with the edge.

Returns: True if the point is on the edge but not equal p1 or p2.

A point is on a edge if it is on (or at least closer than a grid point to) the edge.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

crossed_by?

Signature: *[const]* bool **crossed_by?** (const Edge e)

Description: Checks, if the line given by self is crossed by the edge e

e: The edge representing the line that the edge must be crossing.

self if considered an infinite line. This predicate renders true if the edge e is cut by this line. In other words: this method returns true if e.p1 is in one semispace of self while e.p2 is in the other or one of them is exactly on self.

crossing_point

Signature: *[const]* Point **crossing_point** (const Edge e)

Description: Returns the crossing point on two edges.

e: The edge representing the line that self must be crossing.

Returns: The point where self crosses the line given by "e".

This method delivers the point where the given line (self) crosses the edge given by the argument "e". self is considered infinitely long and is required to cut through the edge "e". If self does not cut this line, the result is undefined. See crossed_by? for a description of the crossing predicate.

This method has been introduced in version 0.19.

cut_point

Signature: *[const]* variant **cut_point** (const Edge e)

Description: Returns the intersection point of the lines through the two edges.

e: The edge to test.

Returns: The point where the lines intersect.

This method delivers the intersection point between the lines through the two edges. If the lines are parallel and do not intersect, the result will be nil. In contrast to intersection_point, this method will regard the edges as infinitely extended and intersection is not confined to the edge span.

This method has been introduced in version 0.27.1.

d

Signature: *[const]* Vector **d**

Description: Gets the edge extension as a vector.

This method is equivalent to p2 - p1. This method has been introduced in version 0.26.2.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

distance

Signature: *[const]* int **distance** (const Point p)

Description: Gets the distance of the point from the line through the edge.

p:	The point to test.
Returns:	The distance

Returns the distance between the edge and the point. The distance is signed which is negative if the point is to the "right" of the edge and positive if the point is to the "left". The distance is measured by projecting the point onto the line through the edge. If the edge is degenerated, the distance is not defined.

This method considers the edge to define an infinite line running through it. distance returns the distance of 'p' to this line. A similar method is euclidian_distance, but the latter regards the edge a finite set of points between the endpoints.

distance_abs

Signature: *[const]* unsigned int **distance_abs** (const Point p)

Description: Absolute distance between the edge and a point.

p:	The point to test.
Returns:	The distance

Returns the distance between the edge and the point.

dup

Signature: *[const]* new Edge ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

dx

Signature: *[const]* int **dx**

Description: The horizontal extend of the edge.

dx_abs

Signature: *[const]* unsigned int **dx_abs**

Description: The absolute value of the horizontal extend of the edge.

dy

Signature: *[const]* int **dy**

Description: The vertical extend of the edge.

dy_abs

Signature: *[const]* unsigned int **dy_abs**

Description: The absolute value of the vertical extend of the edge.

enlarge

Signature: Edge **enlarge** (const Vector p)

Description: Enlarges the edge.

p: The distance to move the edge points.

Returns: The enlarged edge.

Enlarges the edge by the given distance and returns the enlarged edge. The edge is overwritten. Enlargement means that the first point is shifted by -p, the second by p.

enlarged

Signature: *[const]* Edge **enlarged** (const Vector p)

Description: Returns the enlarged edge (does not modify self)

p: The distance to move the edge points.

Returns: The enlarged edge.

Enlarges the edge by the given offset and returns the enlarged edge. The edge is not modified. Enlargement means that the first point is shifted by -p, the second by p.

euclidian_distance

Signature: unsigned int **euclidian_distance** (const Point p)

Description: Gets the distance of the point from the the edge.

p: The point to test.

Returns: The distance

Returns the minimum distance of the point to any point on the edge. Unlike distance, the edge is considered a finite set of points between the endpoints. The result is also not signed like it is the case for distance.

This method has been introduced in version 0.28.14.

extend

Signature: Edge **extend** (int d)

Description: Extends the edge (modifies self)

d: The distance by which to shift the end points.

Returns: The extended edge (self).

Extends the edge by the given distance and returns the extended edge. The edge is not modified. Extending means that the first point is shifted by -d along the edge, the second by d. The length of the edge will increase by 2*d.

extended is a version that does not modify self but returns the extended edges.

This method has been introduced in version 0.23.

extended

Signature: *[const]* Edge **extended** (int d)

Description: Returns the extended edge (does not modify self)

d: The distance by which to shift the end points.

Returns: The extended edge.

Extends the edge by the given distance and returns the extended edge. The edge is not modified. Extending means that the first point is shifted by -d along the edge, the second by d. The length of the edge will increase by 2*d.

extend is a version that modifies self (in-place).

This method has been introduced in version 0.23.

from_dedge

Signature: *[static]* new Edge ptr **from_dedge** (const DEdge dedge)

Description: Creates an integer coordinate edge from a floating-point coordinate edge

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dedge'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new Edge ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given edge. This method enables edges as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

intersect?

Signature: *[const]* bool **intersect?** (const Edge e)

Description: Intersection test.

e: The edge to test.

Use of this method is deprecated. Use intersects? instead

Returns true if the edges intersect. Two edges intersect if they share at least one point. If the edges coincide, they also intersect. If one of the edges is degenerate (both points are identical), that point is required to sit exaclty on the other edge. If both edges are degenerate, their points are required to be identical.

The 'intersects' (with an 's') synonym has been introduced in version 0.28.12.

intersection_point

Signature: *[const]* variant **intersection_point** (const Edge e)

Description: Returns the intersection point of two edges.

e: The edge to test.

Returns: The point where the edges intersect.

This method delivers the intersection point. If the edges do not intersect, the result will be nil.

This method has been introduced in version 0.19. From version 0.26.2, this method will return nil in case of non-intersection.

intersects?

Signature: *[const]* bool **intersects?** (const Edge e)

Description: Intersection test.

e: The edge to test.

Returns true if the edges intersect. Two edges intersect if they share at least one point. If the edges coincide, they also intersect. If one of the edges is degenerate (both points are identical), that point is required to sit exaclty on the other edge. If both edges are degenerate, their points are required to be identical.

The 'intersects' (with an 's') synonym has been introduced in version 0.28.12.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_degenerate?

Signature: *[const]* bool **is_degenerate?**

Description: Test for degenerated edge

An edge is degenerate, if both end and start point are identical.

is_parallel?

Signature: *[const]* bool **is_parallel?** (const Edge e)

Description: Test for being parallel

e: The edge to test against

Returns: True if both edges are parallel

length

Signature: *[const]* unsigned int **length**

Description: The length of the edge

move

(1) Signature: Edge **move** (const Vector p)

Description: Moves the edge.

p: The distance to move the edge.

Returns: The moved edge.

Moves the edge by the given offset and returns the moved edge. The edge is overwritten.

(2) Signature: Edge **move** (int dx, int dy)

Description: Moves the edge.

dx: The x distance to move the edge.

dy: The y distance to move the edge.

Returns: The moved edge.

Moves the edge by the given offset and returns the moved edge. The edge is overwritten.

This version has been added in version 0.23.

moved

(1) Signature: *[const]* Edge **moved** (const Vector p)

Description: Returns the moved edge (does not modify self)

p: The distance to move the edge.

Returns: The moved edge.

Moves the edge by the given offset and returns the moved edge. The edge is not modified.

(2) Signature: *[const]* Edge **moved** (int dx, int dy)

Description: Returns the moved edge (does not modify self)

dx: The x distance to move the edge.

dy: The y distance to move the edge.

Returns: The moved edge.

Moves the edge by the given offset and returns the moved edge. The edge is not modified.

This version has been added in version 0.23.

new

(1) Signature: *[static]* new Edge ptr **new** (const DEdge dedge)

Description: Creates an integer coordinate edge from a floating-point coordinate edge

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dedge'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Edge ptr **new**

Description: Default constructor: creates a degenerated edge 0,0 to 0,0

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Edge ptr **new** (int x1, int y1, int x2, int y2)

Description: Constructor with two coordinates given as single values

Two points are given to create a new edge.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Edge ptr **new** (const Point p1, const Point p2)

Description: Constructor with two points

Two points are given to create a new edge.

Python specific notes:
This method is the default initializer of the object.

new_pp

Signature: *[static]* new Edge ptr **new_pp** (const Point p1, const Point p2)

Description: Constructor with two points

Use of this method is deprecated. Use new instead

Two points are given to create a new edge.

Python specific notes:
This method is the default initializer of the object.

new_xyxy

Signature: *[static]* new Edge ptr **new_xyxy** (int x1, int y1, int x2, int y2)

Description: Constructor with two coordinates given as single values

Use of this method is deprecated. Use new instead

Two points are given to create a new edge.

Python specific notes:
This method is the default initializer of the object.

ortho_length

Signature: *[const]* unsigned int **ortho_length**

Description: The orthogonal length of the edge ("manhattan-length")

Returns: The orthogonal length (abs(dx)+abs(dy))

p1

Signature: *[const]* Point **p1**

Description: The first point.

Python specific notes:
The object exposes a readable attribute 'p1'. This is the getter.

p1=

Signature: void **p1=** (const Point point)

Description: Sets the first point.

This method has been added in version 0.23.

Python specific notes:
The object exposes a writable attribute 'p1'. This is the setter.

p2

Signature: *[const]* Point **p2**

Description: The second point.

Python specific notes:
The object exposes a readable attribute 'p2'. This is the getter.

p2=

Signature: void **p2=** (const Point point)

Description: Sets the second point.

This method has been added in version 0.23.

Python specific notes:
The object exposes a writable attribute 'p2'. This is the setter.

shift

Signature: Edge **shift** (int d)

Description: Shifts the edge (modifies self)

d: The distance by which to shift the edge.

Returns: The shifted edge (self).

Shifts the edge by the given distance and returns the shifted edge. The edge is not modified. Shifting by a positive value will produce an edge which is shifted by d to the left. Shifting by a negative value will produce an edge which is shifted by d to the right.

shifted is a version that does not modify self but returns the extended edges.

This method has been introduced in version 0.23.

shifted

Signature: *[const]* Edge **shifted** (int d)

Description: Returns the shifted edge (does not modify self)

d: The distance by which to shift the edge.

Returns: The shifted edge.

Shifts the edge by the given distance and returns the shifted edge. The edge is not modified. Shifting by a positive value will produce an edge which is shifted by d to the left. Shifting by a negative value will produce an edge which is shifted by d to the right.

shift is a version that modifies self (in-place).

This method has been introduced in version 0.23.

side_of

Signature: *[const]* int **side_of** (const Point p)

Description: Indicates at which side the point is located relative to the edge.

p: The point to test.

Returns: The side value

Returns 1 if the point is "left" of the edge, 0 if on and -1 if the point is "right" of the edge.

sq_length

Signature: *[const]* long **sq_length**

Description: The square of the length of the edge

swap_points

Signature: Edge **swap_points**

Description: Swap the points of the edge

This version modifies self. A version that does not modify self is swapped_points. Swapping the points basically reverses the direction of the edge.

This method has been introduced in version 0.23.

swapped_points

Signature: *[const]* Edge **swapped_points**

Description: Returns an edge in which both points are swapped

Swapping the points basically reverses the direction of the edge.

This method has been introduced in version 0.23.

to_dtype

Signature: *[const]* DEdge **to_dtype** (double dbu = 1)

Description: Converts the edge to a floating-point coordinate edge

The database unit can be specified to translate the integer-coordinate edge into a floating-point coordinate edge in micron units. The database unit is basically a scaling factor.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: Returns a string representing the edge

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

transformed

(1) Signature: *[const]* Edge **transformed** (const ICplxTrans t)

Description: Transform the edge.

t: The transformation to apply.

Returns: The transformed edge (in this case an integer coordinate edge).

Transforms the edge with the given complex transformation. Does not modify the edge but returns the transformed edge.

This method has been introduced in version 0.18.

(2) Signature: *[const]* Edge **transformed** (const Trans t)

Description: Transform the edge.

t: The transformation to apply.

Returns: The transformed edge.

Transforms the edge with the given transformation. Does not modify the edge but returns the transformed edge.

(3) Signature: *[const]* DEdge **transformed** (const CplxTrans t)

Description: Transform the edge.

t: The transformation to apply.

Returns: The transformed edge.

Transforms the edge with the given complex transformation. Does not modify the edge but returns the transformed edge.

transformed_cplx

Signature: *[const]* DEdge **transformed_cplx** (const CplxTrans t)

Description: Transform the edge.

t: The transformation to apply.

Returns: The transformed edge.

Use of this method is deprecated. Use transformed instead

Transforms the edge with the given complex transformation. Does not modify the edge but returns the transformed edge.

x1

Signature: *[const]* int **x1**

Description: Shortcut for p1.x

Python specific notes:
The object exposes a readable attribute 'x1'. This is the getter.

x1=

Signature: void **x1=** (int coord)

Description: Sets p1.x

This method has been added in version 0.23.

Python specific notes:
The object exposes a writable attribute 'x1'. This is the setter.

x2

Signature: *[const]* int **x2**

Description: Shortcut for p2.x

Python specific notes:

	The object exposes a readable attribute 'x2'. This is the getter.
x2=	**Signature**: void **x2=** (int coord) **Description**: Sets p2.x This method has been added in version 0.23. **Python specific notes:** The object exposes a writable attribute 'x2'. This is the setter.
y1	**Signature**: *[const]* int **y1** **Description**: Shortcut for p1.y **Python specific notes:** The object exposes a readable attribute 'y1'. This is the getter.
y1=	**Signature**: void **y1=** (int coord) **Description**: Sets p1.y This method has been added in version 0.23. **Python specific notes:** The object exposes a writable attribute 'y1'. This is the setter.
y2	**Signature**: *[const]* int **y2** **Description**: Shortcut for p2.y **Python specific notes:** The object exposes a readable attribute 'y2'. This is the getter.
y2=	**Signature**: void **y2=** (int coord) **Description**: Sets p2.y This method has been added in version 0.23. **Python specific notes:** The object exposes a writable attribute 'y2'. This is the setter.

4.34. API reference - Class DEdge

Notation used in Ruby API documentation

Module: db

Description: An edge class

An edge is a connection between points, usually participating in a larger context such as a polygon. An edge has a defined direction (from p1 to p2). Edges play a role in the database as parts of polygons and to describe a line through both points. The Edge object is also used inside the boolean processor (EdgeProcessor). Although supported, edges are rarely used as individual database objects.

See The Database API for more details about the database objects like the Edge class.

Public constructors

new DEdge ptr	**new**	(const Edge edge)	Creates a floating-point coordinate edge from an integer coordinate edge
new DEdge ptr	**new**		Default constructor: creates a degenerated edge 0,0 to 0,0
new DEdge ptr	**new**	(double x1, double y1, double x2, double y2)	Constructor with two coordinates given as single values
new DEdge ptr	**new**	(const DPoint p1, const DPoint p2)	Constructor with two points

Public methods

[const]	bool	**!=**	(const DEdge e)	Inequality test
[const]	DEdge	*****	(double scale_factor)	Scale edge
[const]	bool	**<**	(const DEdge e)	Less operator
[const]	bool	**==**	(const DEdge e)	Equality test
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const DEdge other)	Assigns another object to self

[const]	DBox	**bbox**		Return the bounding box of the edge.
[const]	variant	**clipped**	(const DBox box)	Returns the edge clipped at the given box
[const]	variant	**clipped_line**	(const DBox box)	Returns the line through the edge clipped at the given box
[const]	bool	**coincident?**	(const DEdge e)	Coincidence check.
[const]	bool	**contains?**	(const DPoint p)	Tests whether a point is on an edge.
[const]	bool	**contains_excl?**	(const DPoint p)	Tests whether a point is on an edge excluding the endpoints.
[const]	bool	**crossed_by?**	(const DEdge e)	Checks, if the line given by self is crossed by the edge e
[const]	DPoint	**crossing_point**	(const DEdge e)	Returns the crossing point on two edges.
[const]	variant	**cut_point**	(const DEdge e)	Returns the intersection point of the lines through the two edges.
[const]	DVector	**d**		Gets the edge extension as a vector.
[const]	double	**distance**	(const DPoint p)	Gets the distance of the point from the line through the edge.
[const]	double	**distance_abs**	(const DPoint p)	Absolute distance between the edge and a point.
[const]	new DEdge ptr	**dup**		Creates a copy of self
[const]	double	**dx**		The horizontal extend of the edge.
[const]	double	**dx_abs**		The absolute value of the horizontal extend of the edge.
[const]	double	**dy**		The vertical extend of the edge.
[const]	double	**dy_abs**		The absolute value of the vertical extend of the edge.
	DEdge	**enlarge**	(const DVector p)	Enlarges the edge.
[const]	DEdge	**enlarged**	(const DVector p)	Returns the enlarged edge (does not modify self)
	double	**euclidian_distance**	(const DPoint p)	Gets the distance of the point from the the edge.
	DEdge	**extend**	(double d)	Extends the edge (modifies self)
[const]	DEdge	**extended**	(double d)	Returns the extended edge (does not modify self)
[const]	unsigned long	**hash**		Computes a hash value

[const]	variant	**intersection_point**	(const DEdge e)	Returns the intersection point of two edges.
[const]	bool	**intersects?**	(const DEdge e)	Intersection test.
[const]	bool	**is_degenerate?**		Test for degenerated edge
[const]	bool	**is_parallel?**	(const DEdge e)	Test for being parallel
[const]	double	**length**		The length of the edge
	DEdge	**move**	(const DVector p)	Moves the edge.
	DEdge	**move**	(double dx, double dy)	Moves the edge.
[const]	DEdge	**moved**	(const DVector p)	Returns the moved edge (does not modify self)
[const]	DEdge	**moved**	(double dx, double dy)	Returns the moved edge (does not modify self)
[const]	double	**ortho_length**		The orthogonal length of the edge ("manhattan-length")
[const]	DPoint	**p1**		The first point.
	void	**p1=**	(const DPoint point)	Sets the first point.
[const]	DPoint	**p2**		The second point.
	void	**p2=**	(const DPoint point)	Sets the second point.
	DEdge	**shift**	(double d)	Shifts the edge (modifies self)
[const]	DEdge	**shifted**	(double d)	Returns the shifted edge (does not modify self)
[const]	int	**side_of**	(const DPoint p)	Indicates at which side the point is located relative to the edge.
[const]	double	**sq_length**		The square of the length of the edge
	DEdge	**swap_points**		Swap the points of the edge
[const]	DEdge	**swapped_points**		Returns an edge in which both points are swapped
[const]	Edge	**to_itype**	(double dbu = 1)	Converts the edge to an integer coordinate edge
[const]	string	**to_s**	(double dbu = 0)	Returns a string representing the edge
[const]	Edge	**transformed**	(const VCplxTrans t)	Transforms the edge with the given complex transformation
[const]	DEdge	**transformed**	(const DTrans t)	Transform the edge.

[const]	DEdge	**transformed**	(const DCplxTrans t)	Transform the edge.
[const]	double	**x1**		Shortcut for p1.x
	void	**x1=**	(double coord)	Sets p1.x
[const]	double	**x2**		Shortcut for p2.x
	void	**x2=**	(double coord)	Sets p2.x
[const]	double	**y1**		Shortcut for p1.y
	void	**y1=**	(double coord)	Sets p1.y
[const]	double	**y2**		Shortcut for p2.y
	void	**y2=**	(double coord)	Sets p2.y

Public static methods and constants

new DEdge ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new DEdge ptr	**from_iedge**	(const Edge edge)	Use of this method is deprecated. Use new instead
[const]	bool	**intersect?**	(const DEdge e)	Use of this method is deprecated. Use intersects? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[static]	new DEdge ptr	**new_pp**	(const DPoint p1, const DPoint p2)	Use of this method is deprecated. Use new instead
[static]	new DEdge ptr	**new_xyxy**	(double x1, double y1, double x2, double y2)	Use of this method is deprecated. Use new instead
[const]	DEdge	**transformed_cplx**	(const DCplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

!=

Signature: *[const]* bool **!=** (const DEdge e)

Description: Inequality test

e: The object to compare against

*****

Signature: *[const]* DEdge ***** (double scale_factor)

Description: Scale edge

scale_factor: The scaling factor

Returns: The scaled edge

The * operator scales self with the given factor.

This method has been introduced in version 0.22.

Python specific notes:
This method also implements '__rmul__'.

<

Signature: *[const]* bool **<** (const DEdge e)

Description: Less operator

e: The object to compare against

Returns: True, if the edge is 'less' as the other edge with respect to first and second point

==

Signature: *[const]* bool **==** (const DEdge e)

Description: Equality test

e: The object to compare against

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const DEdge other)

Description: Assigns another object to self

bbox

Signature: *[const]* DBox **bbox**

Description: Return the bounding box of the edge.

clipped

Signature: *[const]* variant **clipped** (const DBox box)

Description: Returns the edge clipped at the given box

box:	The clip box.
Returns:	The clipped edge or nil if the edge does not intersect with the box.

This method has been introduced in version 0.26.2.

clipped_line

Signature: *[const]* variant **clipped_line** (const DBox box)

Description: Returns the line through the edge clipped at the given box

box:	The clip box.
Returns:	The part of the line through the box or nil if the line does not intersect with the box.

In contrast to clipped, this method will consider the edge extended infinitely (a "line"). The returned edge will be the part of this line going through the box.

This method has been introduced in version 0.26.2.

coincident?

Signature: *[const]* bool **coincident?** (const DEdge e)

Description: Coincidence check.

e:	the edge to test with

Returns: True if the edges are coincident.

Checks whether a edge is coincident with another edge. Coincidence is defined by being parallel and that at least one point of one edge is on the other edge.

contains?

Signature: *[const]* bool **contains?** (const DPoint p)

Description: Tests whether a point is on an edge.

p: The point to test with the edge.

Returns: True if the point is on the edge.

A point is on a edge if it is on (or at least closer than a grid point to) the edge.

contains_excl?

Signature: *[const]* bool **contains_excl?** (const DPoint p)

Description: Tests whether a point is on an edge excluding the endpoints.

p: The point to test with the edge.

Returns: True if the point is on the edge but not equal p1 or p2.

A point is on a edge if it is on (or at least closer than a grid point to) the edge.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

crossed_by?

Signature: *[const]* bool **crossed_by?** (const DEdge e)

Description: Checks, if the line given by self is crossed by the edge e

e: The edge representing the line that the edge must be crossing.

self if considered an infinite line. This predicate renders true if the edge e is cut by this line. In other words: this method returns true if e.p1 is in one semispace of self while e.p2 is in the other or one of them is exactly on self.

crossing_point

Signature: *[const]* DPoint **crossing_point** (const DEdge e)

Description: Returns the crossing point on two edges.

e: The edge representing the line that self must be crossing.

Returns: The point where self crosses the line given by "e".

This method delivers the point where the given line (self) crosses the edge given by the argument "e". self is considered infinitely long and is required to cut through the edge "e". If self does not cut this line, the result is undefined. See crossed_by? for a description of the crossing predicate.

This method has been introduced in version 0.19.

cut_point

Signature: *[const]* variant **cut_point** (const DEdge e)

Description: Returns the intersection point of the lines through the two edges.

e: The edge to test.

Returns: The point where the lines intersect.

This method delivers the intersection point between the lines through the two edges. If the lines are parallel and do not intersect, the result will be nil. In contrast to intersection_point, this method will regard the edges as infinitely extended and intersection is not confined to the edge span.

This method has been introduced in version 0.27.1.

d

Signature: *[const]* DVector **d**

Description: Gets the edge extension as a vector.

This method is equivalent to p2 - p1. This method has been introduced in version 0.26.2.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

distance

Signature: *[const]* double **distance** (const DPoint p)

Description: Gets the distance of the point from the line through the edge.

p:	The point to test.
Returns:	The distance

Returns the distance between the edge and the point. The distance is signed which is negative if the point is to the "right" of the edge and positive if the point is to the "left". The distance is measured by projecting the point onto the line through the edge. If the edge is degenerated, the distance is not defined.

This method considers the edge to define an infinite line running through it. distance returns the distance of 'p' to this line. A similar method is euclidian_distance, but the latter regards the edge a finite set of points between the endpoints.

distance_abs

Signature: *[const]* double **distance_abs** (const DPoint p)

Description: Absolute distance between the edge and a point.

p:	The point to test.
Returns:	The distance

Returns the distance between the edge and the point.

dup

Signature: *[const]* new DEdge ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

dx

Signature: *[const]* double **dx**

Description: The horizontal extend of the edge.

dx_abs

Signature: *[const]* double **dx_abs**

Description: The absolute value of the horizontal extend of the edge.

dy

Signature: *[const]* double **dy**

Description: The vertical extend of the edge.

dy_abs

Signature: *[const]* double **dy_abs**

Description: The absolute value of the vertical extend of the edge.

enlarge

Signature: DEdge **enlarge** (const DVector p)

Description: Enlarges the edge.

p: The distance to move the edge points.

Returns: The enlarged edge.

Enlarges the edge by the given distance and returns the enlarged edge. The edge is overwritten. Enlargement means that the first point is shifted by -p, the second by p.

enlarged

Signature: *[const]* DEdge **enlarged** (const DVector p)

Description: Returns the enlarged edge (does not modify self)

p: The distance to move the edge points.

Returns: The enlarged edge.

Enlarges the edge by the given offset and returns the enlarged edge. The edge is not modified. Enlargement means that the first point is shifted by -p, the second by p.

euclidian_distance

Signature: double **euclidian_distance** (const DPoint p)

Description: Gets the distance of the point from the the edge.

p: The point to test.

Returns: The distance

Returns the minimum distance of the point to any point on the edge. Unlike distance, the edge is considered a finite set of points between the endpoints. The result is also not signed like it is the case for distance.

This method has been introduced in version 0.28.14.

extend

Signature: DEdge **extend** (double d)

Description: Extends the edge (modifies self)

d: The distance by which to shift the end points.

Returns: The extended edge (self).

Extends the edge by the given distance and returns the extended edge. The edge is not modified. Extending means that the first point is shifted by -d along the edge, the second by d. The length of the edge will increase by 2*d.

extended is a version that does not modify self but returns the extended edges.

This method has been introduced in version 0.23.

extended

Signature: *[const]* DEdge **extended** (double d)

Description: Returns the extended edge (does not modify self)

d: The distance by which to shift the end points.

Returns: The extended edge.

Extends the edge by the given distance and returns the extended edge. The edge is not modified. Extending means that the first point is shifted by -d along the edge, the second by d. The length of the edge will increase by 2*d.

extend is a version that modifies self (in-place).

This method has been introduced in version 0.23.

from_iedge

Signature: *[static]* new DEdge ptr **from_iedge** (const Edge edge)

Description: Creates a floating-point coordinate edge from an integer coordinate edge

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_iedge'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new DEdge ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given edge. This method enables edges as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

intersect?

Signature: *[const]* bool **intersect?** (const DEdge e)

Description: Intersection test.

e: The edge to test.

Use of this method is deprecated. Use intersects? instead

Returns true if the edges intersect. Two edges intersect if they share at least one point. If the edges coincide, they also intersect. If one of the edges is degenerate (both points are identical), that point is required to sit exaclty on the other edge. If both edges are degenerate, their points are required to be identical.

The 'intersects' (with an 's') synonym has been introduced in version 0.28.12.

intersection_point

Signature: *[const]* variant **intersection_point** (const DEdge e)

Description: Returns the intersection point of two edges.

e: The edge to test.

Returns: The point where the edges intersect.

This method delivers the intersection point. If the edges do not intersect, the result will be nil.

This method has been introduced in version 0.19. From version 0.26.2, this method will return nil in case of non-intersection.

intersects?

Signature: *[const]* bool **intersects?** (const DEdge e)

Description: Intersection test.

e: The edge to test.

Returns true if the edges intersect. Two edges intersect if they share at least one point. If the edges coincide, they also intersect. If one of the edges is degenerate (both points are identical), that point is required to sit exaclty on the other edge. If both edges are degenerate, their points are required to be identical.

The 'intersects' (with an 's') synonym has been introduced in version 0.28.12.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_degenerate?

Signature: *[const]* bool **is_degenerate?**

Description: Test for degenerated edge

An edge is degenerate, if both end and start point are identical.

is_parallel?

Signature: *[const]* bool **is_parallel?** (const DEdge e)

Description: Test for being parallel

e: The edge to test against

Returns: True if both edges are parallel

length

Signature: *[const]* double **length**

Description: The length of the edge

move

(1) Signature: DEdge **move** (const DVector p)

Description: Moves the edge.

p: The distance to move the edge.

Returns: The moved edge.

Moves the edge by the given offset and returns the moved edge. The edge is overwritten.

(2) Signature: DEdge **move** (double dx, double dy)

Description: Moves the edge.

dx: The x distance to move the edge.

dy: The y distance to move the edge.

Returns: The moved edge.

Moves the edge by the given offset and returns the moved edge. The edge is overwritten.

This version has been added in version 0.23.

moved

(1) Signature: *[const]* DEdge **moved** (const DVector p)

Description: Returns the moved edge (does not modify self)

p:	The distance to move the edge.
Returns:	The moved edge.

Moves the edge by the given offset and returns the moved edge. The edge is not modified.

(2) Signature: *[const]* DEdge **moved** (double dx, double dy)

Description: Returns the moved edge (does not modify self)

dx:	The x distance to move the edge.
dy:	The y distance to move the edge.
Returns:	The moved edge.

Moves the edge by the given offset and returns the moved edge. The edge is not modified.

This version has been added in version 0.23.

new

(1) Signature: *[static]* new DEdge ptr **new** (const Edge edge)

Description: Creates a floating-point coordinate edge from an integer coordinate edge

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_iedge'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DEdge ptr **new**

Description: Default constructor: creates a degenerated edge 0,0 to 0,0

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new DEdge ptr **new** (double x1, double y1, double x2, double y2)

Description: Constructor with two coordinates given as single values

Two points are given to create a new edge.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new DEdge ptr **new** (const DPoint p1, const DPoint p2)

Description: Constructor with two points

Two points are given to create a new edge.

Python specific notes:
This method is the default initializer of the object.

new_pp

Signature: *[static]* new DEdge ptr **new_pp** (const DPoint p1, const DPoint p2)

Description: Constructor with two points

Use of this method is deprecated. Use new instead

Two points are given to create a new edge.

Python specific notes:
This method is the default initializer of the object.

new_xyxy

Signature: *[static]* new DEdge ptr **new_xyxy** (double x1, double y1, double x2, double y2)

Description: Constructor with two coordinates given as single values

Use of this method is deprecated. Use new instead

Two points are given to create a new edge.

Python specific notes:
This method is the default initializer of the object.

ortho_length

Signature: *[const]* double **ortho_length**

Description: The orthogonal length of the edge ("manhattan-length")

Returns: The orthogonal length (abs(dx)+abs(dy))

p1

Signature: *[const]* DPoint **p1**

Description: The first point.

Python specific notes:
The object exposes a readable attribute 'p1'. This is the getter.

p1=

Signature: void **p1=** (const DPoint point)

Description: Sets the first point.

This method has been added in version 0.23.

Python specific notes:
The object exposes a writable attribute 'p1'. This is the setter.

p2

Signature: *[const]* DPoint **p2**

Description: The second point.

Python specific notes:
The object exposes a readable attribute 'p2'. This is the getter.

p2=

Signature: void **p2=** (const DPoint point)

Description: Sets the second point.

This method has been added in version 0.23.

Python specific notes:
The object exposes a writable attribute 'p2'. This is the setter.

shift

Signature: DEdge **shift** (double d)

Description: Shifts the edge (modifies self)

d: The distance by which to shift the edge.

Returns: The shifted edge (self).

Shifts the edge by the given distance and returns the shifted edge. The edge is not modified. Shifting by a positive value will produce an edge which is shifted by d to the left. Shifting by a negative value will produce an edge which is shifted by d to the right.

shifted is a version that does not modify self but returns the extended edges.

This method has been introduced in version 0.23.

shifted

Signature: *[const]* DEdge **shifted** (double d)

Description: Returns the shifted edge (does not modify self)

d: The distance by which to shift the edge.

Returns: The shifted edge.

Shifts the edge by the given distance and returns the shifted edge. The edge is not modified. Shifting by a positive value will produce an edge which is shifted by d to the left. Shifting by a negative value will produce an edge which is shifted by d to the right.

shift is a version that modifies self (in-place).

This method has been introduced in version 0.23.

side_of

Signature: *[const]* int **side_of** (const DPoint p)

Description: Indicates at which side the point is located relative to the edge.

p: The point to test.

Returns: The side value

Returns 1 if the point is "left" of the edge, 0 if on and -1 if the point is "right" of the edge.

sq_length

Signature: *[const]* double **sq_length**

Description: The square of the length of the edge

swap_points

Signature: DEdge **swap_points**

Description: Swap the points of the edge

This version modifies self. A version that does not modify self is swapped_points. Swapping the points basically reverses the direction of the edge.

This method has been introduced in version 0.23.

swapped_points

Signature: *[const]* DEdge **swapped_points**

Description: Returns an edge in which both points are swapped

Swapping the points basically reverses the direction of the edge.

This method has been introduced in version 0.23.

to_itype

Signature: *[const]* Edge **to_itype** (double dbu = 1)

Description: Converts the edge to an integer coordinate edge

The database unit can be specified to translate the floating-point coordinate edge in micron units to an integer-coordinate edge in database units. The edges coordinates will be divided by the database unit.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: Returns a string representing the edge

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:

This method is also available as 'str(object)'.

transformed

(1) Signature: *[const]* Edge **transformed** (const VCplxTrans t)

Description: Transforms the edge with the given complex transformation

t:	The magnifying transformation to apply
Returns:	The transformed edge (in this case an integer coordinate edge)

This method has been introduced in version 0.25.

(2) Signature: *[const]* DEdge **transformed** (const DTrans t)

Description: Transform the edge.

t:	The transformation to apply.
Returns:	The transformed edge.

Transforms the edge with the given transformation. Does not modify the edge but returns the transformed edge.

(3) Signature: *[const]* DEdge **transformed** (const DCplxTrans t)

Description: Transform the edge.

t:	The transformation to apply.
Returns:	The transformed edge.

Transforms the edge with the given complex transformation. Does not modify the edge but returns the transformed edge.

transformed_cplx

Signature: *[const]* DEdge **transformed_cplx** (const DCplxTrans t)

Description: Transform the edge.

t:	The transformation to apply.
Returns:	The transformed edge.

Use of this method is deprecated. Use transformed instead

Transforms the edge with the given complex transformation. Does not modify the edge but returns the transformed edge.

x1

Signature: *[const]* double **x1**

Description: Shortcut for p1.x

Python specific notes:
The object exposes a readable attribute 'x1'. This is the getter.

x1=

Signature: void **x1=** (double coord)

Description: Sets p1.x

This method has been added in version 0.23.

Python specific notes:
The object exposes a writable attribute 'x1'. This is the setter.

x2

Signature: *[const]* double **x2**

Description: Shortcut for p2.x

Python specific notes:

	The object exposes a readable attribute 'x2'. This is the getter.
x2=	**Signature**: void **x2=** (double coord) **Description**: Sets p2.x This method has been added in version 0.23. **Python specific notes:** The object exposes a writable attribute 'x2'. This is the setter.
y1	**Signature**: *[const]* double **y1** **Description**: Shortcut for p1.y **Python specific notes:** The object exposes a readable attribute 'y1'. This is the getter.
y1=	**Signature**: void **y1=** (double coord) **Description**: Sets p1.y This method has been added in version 0.23. **Python specific notes:** The object exposes a writable attribute 'y1'. This is the setter.
y2	**Signature**: *[const]* double **y2** **Description**: Shortcut for p2.y **Python specific notes:** The object exposes a readable attribute 'y2'. This is the getter.
y2=	**Signature**: void **y2=** (double coord) **Description**: Sets p2.y This method has been added in version 0.23. **Python specific notes:** The object exposes a writable attribute 'y2'. This is the setter.

4.35. API reference - Class EdgePair

Notation used in Ruby API documentation

Module: db

Description: An edge pair (a pair of two edges)

Edge pairs are objects representing two edges or parts of edges. They play a role mainly in the context of DRC functions, where they specify a DRC violation by connecting two edges which violate the condition checked. Within the framework of polygon and edge collections which provide DRC functionality, edges pairs are used in the form of edge pair collections (EdgePairs).

Edge pairs basically consist of two edges, called first and second. If created by a two-layer DRC function, the first edge will correspond to edges from the first layer and the second to edges from the second layer.

This class has been introduced in version 0.23.

Public constructors

new EdgePair ptr	**new**	(const DEdgePair dedge_pair)	Creates an integer coordinate edge pair from a floating-point coordinate edge pair
new EdgePair ptr	**new**		Default constructor
new EdgePair ptr	**new**	(const Edge first, const Edge second, bool symmetric = false)	Constructor from two edges

Public methods

[const]	bool	**!=**	(const EdgePair box)	Inequality
[const]	bool	**<**	(const EdgePair box)	Less operator
[const]	bool	**==**	(const EdgePair box)	Equality
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	long	**area**		Gets the area between the edges of the edge pair
	void	**assign**	(const EdgePair other)	Assigns another object to self

[const]	Box	**bbox**		Gets the bounding box of the edge pair
[const]	unsigned int	**distance**		Gets the distance of the edges in the edge pair
[const]	new EdgePair ptr	**dup**		Creates a copy of self
[const]	Edge	**first**		Gets the first edge
	void	**first=**	(const Edge edge)	Sets the first edge
[const]	Edge	**greater**		Gets the 'greater' edge for symmetric edge pairs
[const]	unsigned long	**hash**		Computes a hash value
[const]	Edge	**lesser**		Gets the 'lesser' edge for symmetric edge pairs
[const]	EdgePair	**normalized**		Normalizes the edge pair
[const]	unsigned long	**perimeter**		Gets the perimeter of the edge pair
[const]	Polygon	**polygon**	(int e)	Convert an edge pair to a polygon
[const]	Edge	**second**		Gets the second edge
	void	**second=**	(const Edge edge)	Sets the second edge
[const]	SimplePolygon	**simple_polygon**	(int e)	Convert an edge pair to a simple polygon
	void	**symmetric=**	(bool flag)	Sets a value indicating whether the edge pair is symmetric
[const]	bool	**symmetric?**		Returns a value indicating whether the edge pair is symmetric
[const]	DEdgePair	**to_dtype**	(double dbu = 1)	Converts the edge pair to a floating-point coordinate edge pair
[const]	string	**to_s**	(double dbu = 0)	Returns a string representing the edge pair
[const]	EdgePair	**transformed**	(const ICplxTrans t)	Returns the transformed edge pair
[const]	EdgePair	**transformed**	(const Trans t)	Returns the transformed pair
[const]	DEdgePair	**transformed**	(const CplxTrans t)	Returns the transformed edge pair

Public static methods and constants

new EdgePair ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const EdgePair box)

Description: Inequality

Returns true, if this edge pair and the given one are not equal

This method has been introduced in version 0.25.

<

Signature: *[const]* bool **<** (const EdgePair box)

Description: Less operator

Returns true, if this edge pair is 'less' with respect to first and second edge

This method has been introduced in version 0.25.

==

Signature: *[const]* bool **==** (const EdgePair box)

Description: Equality

Returns true, if this edge pair and the given one are equal

This method has been introduced in version 0.25.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

area

Signature: *[const]* long **area**

Description: Gets the area between the edges of the edge pair

This attribute has been introduced in version 0.28.

assign

Signature: void **assign** (const EdgePair other)

Description: Assigns another object to self

bbox

Signature: *[const]* Box **bbox**

Description: Gets the bounding box of the edge pair

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

distance

Signature: *[const]* unsigned int **distance**

Description: Gets the distance of the edges in the edge pair

The distance between the two edges is defined as the minimum distance between any two points on the two edges.

This attribute has been introduced in version 0.28.14.

dup

Signature: *[const]* new EdgePair ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

first

Signature: *[const]* Edge **first**

Description: Gets the first edge

Python specific notes:
The object exposes a readable attribute 'first'. This is the getter.

first=

Signature: void **first=** (const Edge edge)

Description: Sets the first edge

Python specific notes:
The object exposes a writable attribute 'first'. This is the setter.

from_s

Signature: *[static]* new EdgePair ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

greater

Signature: *[const]* Edge **greater**

Description: Gets the 'greater' edge for symmetric edge pairs

As first and second edges are commutable for symmetric edge pairs (see symmetric?), this accessor allows retrieving a 'second' edge in a way independent on the actual assignment.

This read-only attribute has been introduced in version 0.27.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given edge pair. This method enables edge pairs as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

lesser

Signature: *[const]* Edge **lesser**

Description: Gets the 'lesser' edge for symmetric edge pairs

As first and second edges are commutable for symmetric edge pairs (see symmetric?), this accessor allows retrieving a 'first' edge in a way independent on the actual assignment.

This read-only attribute has been introduced in version 0.27.

new

(1) Signature: *[static]* new EdgePair ptr **new** (const DEdgePair dedge_pair)

Description: Creates an integer coordinate edge pair from a floating-point coordinate edge pair

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dedge_pair'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new EdgePair ptr **new**

Description: Default constructor

This constructor creates an default edge pair.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new EdgePair ptr **new** (const Edge first, const Edge second, bool symmetric = false)

Description: Constructor from two edges

This constructor creates an edge pair from the two edges given. See symmetric? for a description of this attribute.

Python specific notes:
This method is the default initializer of the object.

normalized

Signature: *[const]* EdgePair **normalized**

Description: Normalizes the edge pair

This method normalized the edge pair such that when connecting the edges at their start and end points a closed loop is formed which is oriented clockwise. To achieve this, the points of the first and/or first and second edge are swapped. Normalization is a first step recommended before converting an edge pair to a polygon, because that way the polygons won't be self-overlapping and the enlargement parameter is applied properly.

perimeter

Signature: *[const]* unsigned long **perimeter**

Description: Gets the perimeter of the edge pair

The perimeter is defined as the sum of the lengths of both edges ('active perimeter').

This attribute has been introduced in version 0.28.

polygon

Signature: *[const]* Polygon **polygon** (int e)

Description: Convert an edge pair to a polygon

e: The enlargement (set to zero for exact representation)

The polygon is formed by connecting the end and start points of the edges. It is recommended to use normalized before converting the edge pair to a polygon.

The enlargement parameter applies the specified enlargement parallel and perpendicular to the edges. Basically this introduces a bias which blows up edge pairs by the specified amount. That parameter is useful to convert degenerated edge pairs to valid polygons, i.e. edge pairs with coincident edges and edge pairs consisting of two point-like edges.

Another version for converting edge pairs to simple polygons is simple_polygon which renders a SimplePolygon object.

second

Signature: *[const]* Edge **second**

Description: Gets the second edge

Python specific notes:
The object exposes a readable attribute 'second'. This is the getter.

second=

Signature: void **second=** (const Edge edge)

Description: Sets the second edge

Python specific notes:
The object exposes a writable attribute 'second'. This is the setter.

simple_polygon

Signature: *[const]* SimplePolygon **simple_polygon** (int e)

Description: Convert an edge pair to a simple polygon

e: The enlargement (set to zero for exact representation)

The polygon is formed by connecting the end and start points of the edges. It is recommended to use normalized before converting the edge pair to a polygon.

The enlargement parameter applies the specified enlargement parallel and perpendicular to the edges. Basically this introduces a bias which blows up edge pairs by the specified amount. That parameter is useful to convert degenerated edge pairs to valid polygons, i.e. edge pairs with coincident edges and edge pairs consisting of two point-like edges.

Another version for converting edge pairs to polygons is polygon which renders a Polygon object.

symmetric=

Signature: void **symmetric=** (bool flag)

Description: Sets a value indicating whether the edge pair is symmetric

See symmetric? for a description of this attribute.

Symmetric edge pairs have been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'symmetric'. This is the setter.

symmetric?

Signature: *[const]* bool **symmetric?**

Description: Returns a value indicating whether the edge pair is symmetric

For symmetric edge pairs, the edges are commutable. Specifically, a symmetric edge pair with (e1,e2) is identical to (e2,e1). Symmetric edge pairs are generated by some checks for which there is no directed error marker (width, space, notch, isolated).

Symmetric edge pairs have been introduced in version 0.27.

Python specific notes:

The object exposes a readable attribute 'symmetric'. This is the getter.

to_dtype

Signature: *[const]* DEdgePair **to_dtype** (double dbu = 1)

Description: Converts the edge pair to a floating-point coordinate edge pair

The database unit can be specified to translate the integer-coordinate edge pair into a floating-point coordinate edge pair in micron units. The database unit is basically a scaling factor.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: Returns a string representing the edge pair

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

transformed

(1) Signature: *[const]* EdgePair **transformed** (const ICplxTrans t)

Description: Returns the transformed edge pair

t:	The transformation to apply.
Returns:	The transformed edge pair (in this case an integer coordinate edge pair).

Transforms the edge pair with the given complex transformation. Does not modify the edge pair but returns the transformed edge.

(2) Signature: *[const]* EdgePair **transformed** (const Trans t)

Description: Returns the transformed pair

t:	The transformation to apply.
Returns:	The transformed edge pair

Transforms the edge pair with the given transformation. Does not modify the edge pair but returns the transformed edge.

(3) Signature: *[const]* DEdgePair **transformed** (const CplxTrans t)

Description: Returns the transformed edge pair

t:	The transformation to apply.
Returns:	The transformed edge pair

Transforms the edge pair with the given complex transformation. Does not modify the edge pair but returns the transformed edge.

4.36. API reference - Class DEdgePair

Notation used in Ruby API documentation

Module: db

Description: An edge pair (a pair of two edges)

Edge pairs are objects representing two edges or parts of edges. They play a role mainly in the context of DRC functions, where they specify a DRC violation by connecting two edges which violate the condition checked. Within the framework of polygon and edge collections which provide DRC functionality, edges pairs with integer coordinates (EdgePair type) are used in the form of edge pair collections (EdgePairs).

Edge pairs basically consist of two edges, called first and second. If created by a two-layer DRC function, the first edge will correspond to edges from the first layer and the second to edges from the second layer.

This class has been introduced in version 0.23.

Public constructors

new DEdgePair ptr	**new**	(const EdgePair edge_pair)	Creates a floating-point coordinate edge pair from an integer coordinate edge
new DEdgePair ptr	**new**		Default constructor
new DEdgePair ptr	**new**	(const DEdge first, const DEdge second, bool symmetric = false)	Constructor from two edges

Public methods

[const]	bool	**!=**	(const DEdgePair box)	Inequality
[const]	bool	**<**	(const DEdgePair box)	Less operator
[const]	bool	**==**	(const DEdgePair box)	Equality
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	double	**area**		Gets the area between the edges of the edge pair

	void	**assign**	(const DEdgePair other)	Assigns another object to self
[const]	DBox	**bbox**		Gets the bounding box of the edge pair
[const]	double	**distance**		Gets the distance of the edges in the edge pair
[const]	new DEdgePair ptr	**dup**		Creates a copy of self
[const]	DEdge	**first**		Gets the first edge
	void	**first=**	(const DEdge edge)	Sets the first edge
[const]	DEdge	**greater**		Gets the 'greater' edge for symmetric edge pairs
[const]	unsigned long	**hash**		Computes a hash value
[const]	DEdge	**lesser**		Gets the 'lesser' edge for symmetric edge pairs
[const]	DEdgePair	**normalized**		Normalizes the edge pair
[const]	double	**perimeter**		Gets the perimeter of the edge pair
[const]	DPolygon	**polygon**	(double e)	Convert an edge pair to a polygon
[const]	DEdge	**second**		Gets the second edge
	void	**second=**	(const DEdge edge)	Sets the second edge
[const]	DSimplePolygon	**simple_polygon**	(double e)	Convert an edge pair to a simple polygon
	void	**symmetric=**	(bool flag)	Sets a value indicating whether the edge pair is symmetric
[const]	bool	**symmetric?**		Returns a value indicating whether the edge pair is symmetric
[const]	EdgePair	**to_itype**	(double dbu = 1)	Converts the edge pair to an integer coordinate edge pair
[const]	string	**to_s**	(double dbu = 0)	Returns a string representing the edge pair
[const]	EdgePair	**transformed**	(const VCplxTrans t)	Transforms the edge pair with the given complex transformation
[const]	DEdgePair	**transformed**	(const DTrans t)	Returns the transformed pair
[const]	DEdgePair	**transformed**	(const DCplxTrans t)	Returns the transformed edge pair

Public static methods and constants

new DEdgePair ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const DEdgePair box)

Description: Inequality

Returns true, if this edge pair and the given one are not equal

This method has been introduced in version 0.25.

<

Signature: *[const]* bool **<** (const DEdgePair box)

Description: Less operator

Returns true, if this edge pair is 'less' with respect to first and second edge

This method has been introduced in version 0.25.

==

Signature: *[const]* bool **==** (const DEdgePair box)

Description: Equality

Returns true, if this edge pair and the given one are equal

This method has been introduced in version 0.25.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

area

Signature: *[const]* double **area**

Description: Gets the area between the edges of the edge pair

This attribute has been introduced in version 0.28.

assign

Signature: void **assign** (const DEdgePair other)

Description: Assigns another object to self

bbox

Signature: *[const]* DBox **bbox**

Description: Gets the bounding box of the edge pair

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

distance

Signature: *[const]* double **distance**

Description: Gets the distance of the edges in the edge pair

The distance between the two edges is defined as the minimum distance between any two points on the two edges.

This attribute has been introduced in version 0.28.14.

dup

Signature: *[const]* new DEdgePair ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

first

Signature: *[const]* DEdge **first**

Description: Gets the first edge

Python specific notes:
The object exposes a readable attribute 'first'. This is the getter.

first=

Signature: void **first=** (const DEdge edge)

Description: Sets the first edge

Python specific notes:
The object exposes a writable attribute 'first'. This is the setter.

from_s

Signature: *[static]* new DEdgePair ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

greater

Signature: *[const]* DEdge **greater**

Description: Gets the 'greater' edge for symmetric edge pairs

As first and second edges are commutable for symmetric edge pairs (see symmetric?), this accessor allows retrieving a 'second' edge in a way independent on the actual assignment.

This read-only attribute has been introduced in version 0.27.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given edge pair. This method enables edge pairs as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

lesser

Signature: *[const]* DEdge **lesser**

Description: Gets the 'lesser' edge for symmetric edge pairs

As first and second edges are commutable for symmetric edge pairs (see symmetric?), this accessor allows retrieving a 'first' edge in a way independent on the actual assignment.

This read-only attribute has been introduced in version 0.27.

new

(1) Signature: *[static]* new DEdgePair ptr **new** (const EdgePair edge_pair)

Description: Creates a floating-point coordinate edge pair from an integer coordinate edge

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_iedge_pair'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DEdgePair ptr **new**

Description: Default constructor

This constructor creates an default edge pair.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new DEdgePair ptr **new** (const DEdge first, const DEdge second, bool symmetric = false)

Description: Constructor from two edges

This constructor creates an edge pair from the two edges given. See symmetric? for a description of this attribute.

Python specific notes:
This method is the default initializer of the object.

normalized

Signature: *[const]* DEdgePair **normalized**

Description: Normalizes the edge pair

This method normalized the edge pair such that when connecting the edges at their start and end points a closed loop is formed which is oriented clockwise. To achieve this, the points of the first and/or first and second edge are swapped. Normalization is a first step recommended before converting an edge pair to a polygon, because that way the polygons won't be self-overlapping and the enlargement parameter is applied properly.

perimeter

Signature: *[const]* double **perimeter**

Description: Gets the perimeter of the edge pair

The perimeter is defined as the sum of the lengths of both edges ('active perimeter').

This attribute has been introduced in version 0.28.

polygon

Signature: *[const]* DPolygon **polygon** (double e)

Description: Convert an edge pair to a polygon

e: The enlargement (set to zero for exact representation)

The polygon is formed by connecting the end and start points of the edges. It is recommended to use normalized before converting the edge pair to a polygon.

The enlargement parameter applies the specified enlargement parallel and perpendicular to the edges. Basically this introduces a bias which blows up edge pairs by the specified amount. That parameter is useful to convert degenerated edge pairs to valid polygons, i.e. edge pairs with coincident edges and edge pairs consisting of two point-like edges.

Another version for converting edge pairs to simple polygons is simple_polygon which renders a SimplePolygon object.

second

Signature: *[const]* DEdge **second**

Description: Gets the second edge

Python specific notes:
The object exposes a readable attribute 'second'. This is the getter.

second=

Signature: void **second=** (const DEdge edge)

Description: Sets the second edge

Python specific notes:
The object exposes a writable attribute 'second'. This is the setter.

simple_polygon

Signature: *[const]* DSimplePolygon **simple_polygon** (double e)

Description: Convert an edge pair to a simple polygon

e: The enlargement (set to zero for exact representation)

The polygon is formed by connecting the end and start points of the edges. It is recommended to use normalized before converting the edge pair to a polygon.

The enlargement parameter applies the specified enlargement parallel and perpendicular to the edges. Basically this introduces a bias which blows up edge pairs by the specified amount. That parameter is useful to convert degenerated edge pairs to valid polygons, i.e. edge pairs with coincident edges and edge pairs consisting of two point-like edges.

Another version for converting edge pairs to polygons is polygon which renders a Polygon object.

symmetric=

Signature: void **symmetric=** (bool flag)

Description: Sets a value indicating whether the edge pair is symmetric

See symmetric? for a description of this attribute.

Symmetric edge pairs have been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'symmetric'. This is the setter.

symmetric?

Signature: *[const]* bool **symmetric?**

Description: Returns a value indicating whether the edge pair is symmetric

For symmetric edge pairs, the edges are commutable. Specifically, a symmetric edge pair with (e1,e2) is identical to (e2,e1). Symmetric edge pairs are generated by some checks for which there is no directed error marker (width, space, notch, isolated).

Symmetric edge pairs have been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'symmetric'. This is the getter.

to_itype

Signature: *[const]* EdgePair **to_itype** (double dbu = 1)

Description: Converts the edge pair to an integer coordinate edge pair

The database unit can be specified to translate the floating-point coordinate edge pair in micron units to an integer-coordinate edge pair in database units. The edge pair's' coordinates will be divided by the database unit.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: Returns a string representing the edge pair

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

transformed

(1) Signature: *[const]* EdgePair **transformed** (const VCplxTrans t)

Description: Transforms the edge pair with the given complex transformation

t:	The magnifying transformation to apply
Returns:	The transformed edge pair (in this case an integer coordinate edge pair)

This method has been introduced in version 0.25.

(2) Signature: *[const]* DEdgePair **transformed** (const DTrans t)

Description: Returns the transformed pair

t:	The transformation to apply.
Returns:	The transformed edge pair

Transforms the edge pair with the given transformation. Does not modify the edge pair but returns the transformed edge.

(3) Signature: *[const]* DEdgePair **transformed** (const DCplxTrans t)

Description: Returns the transformed edge pair

t:	The transformation to apply.
Returns:	The transformed edge pair

Transforms the edge pair with the given complex transformation. Does not modify the edge pair but returns the transformed edge.

4.37. API reference - Class EdgePairFilter

Notation used in Ruby API documentation

Module: db

Description: A generic edge pair filter adaptor

EdgePair filters are an efficient way to filter edge pairs from a EdgePairs collection. To apply a filter, derive your own filter class and pass an instance to EdgePairs#filter or EdgePairs#filtered method.

Conceptually, these methods take each edge pair from the collection and present it to the filter's 'selected' method. Based on the result of this evaluation, the edge pair is kept or discarded.

The magic happens when deep mode edge pair collections are involved. In that case, the filter will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the filter behaves. You need to configure the filter by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using the filter.

You can skip this step, but the filter algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

Here is some example that filters edge pairs where the edges are perpendicular:

```
class PerpendicularEdgesFilter < RBA::EdgePairFilter

  # Constructor
  def initialize
    self.is_isotropic_and_scale_invariant   # orientation and scale do not matter
  end

  # Select edge pairs where the edges are perpendicular
  def selected(edge_pair)
    return edge_pair.first.d.sprod_sign(edge_pair.second.d) == 0
  end

end

edge_pairs = ... # some EdgePairs object
perpendicular_only = edge_pairs.filtered(PerpendicularEdgesFilter::new)
```

This class has been introduced in version 0.29.

Public constructors

new EdgePairFilter ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.

	void	_unmanage		Marks the object as no longer owned by the script side.
	void	is_isotropic		Indicates that the filter has isotropic properties
	void	is_isotropic_and_scale_invariant		Indicates that the filter is isotropic and scale invariant
	void	is_scale_invariant		Indicates that the filter is scale invariant
[virtual,const]	bool	selected	(const EdgePair text)	Selects an edge pair
	void	wants_variants=	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	wants_variants?		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) filters are area or perimeter filters. The area or perimeter of a polygon depends on the scale, but not on the orientation of the polygon.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) filter is the square selector. Whether a polygon is a square or not does not depend on the polygon's orientation nor scale.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) filter is the bounding box aspect ratio (height/width) filter. The definition of heigh and width depends on the orientation, but the ratio is independent on scale.

new

Signature: *[static]* new EdgePairFilter ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

selected

Signature: *[virtual,const]* bool **selected** (const EdgePair text)

Description: Selects an edge pair

This method is the actual payload. It needs to be reimplemented in a derived class. It needs to analyze the edge pair and return 'true' if it should be kept and 'false' if it should be discarded.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.38. API reference - Class EdgePairOperator

Notation used in Ruby API documentation

Module: db

Description: A generic edge-pair operator

Edge pair processors are an efficient way to process edge pairs from an edge pair collection. To apply a processor, derive your own operator class and pass an instance to the EdgePairs#processed or EdgePairs#process method.

Conceptually, these methods take each edge pair from the edge pair collection and present it to the operator's 'process' method. The result of this call is a list of zero to many output edge pairs derived from the input edge pair. The output edge pair collection is the sum over all these individual results.

The magic happens when deep mode edge pair collections are involved. In that case, the processor will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the operator behaves. You need to configure the operator by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using it.

You can skip this step, but the processor algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

Here is some example that flips the edge pairs (swaps first and second edge):

```
class FlipEdgePairs < RBA::EdgePairOperator

  # Constructor
  def initialize
    self.is_isotropic_and_scale_invariant   # orientation and scale do not matter
  end

  # Flips the edge pair
  def process(edge_pair)
    return [ RBA::EdgePair::new(edge_pair.second, edge_pair.first) ]
  end

end

edge_pairs = ... # some EdgePairs object
flipped = edge_pairs.processed(FlipEdgePairs::new)
```

This class has been introduced in version 0.29.

Public constructors

new EdgePairOperator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.

	void	_unmanage		Marks the object as no longer owned by the script side.
	void	is_isotropic		Indicates that the filter has isotropic properties
	void	is_isotropic_and_scale_invariant		Indicates that the filter is isotropic and scale invariant
	void	is_scale_invariant		Indicates that the filter is scale invariant
[virtual,const]	EdgePair[]	process	(const EdgePair shape)	Processes a shape
	void	wants_variants=	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	wants_variants?		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) processors are size or shrink operators. Size or shrink is not dependent on orientation unless size or shrink needs to be different in x and y direction.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) processor is the convex decomposition operator. The decomposition of a polygon into convex parts is an operation that is not depending on scale nor orientation.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) processor is the rotation operator. Rotation is not depending on scale, but on the original orientation as mirrored versions need to be rotated differently.

new

Signature: *[static]* new EdgePairOperator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

process

Signature: *[virtual,const]* EdgePair[] **process** (const EdgePair shape)

Description: Processes a shape

This method is the actual payload. It needs to be reimplemented in a derived class. If needs to process the input shape and deliver a list of output shapes. The output list may be empty to entirely discard the input shape. It may also contain more than a single shape. In that case, the number of total shapes may grow during application of the processor.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.39. API reference - Class EdgePairToPolygonOperator

Notation used in Ruby API documentation

Module: db

Description: A generic edge-pair-to-polygon operator

Edge pair processors are an efficient way to process edge pairs from an edge pair collection. To apply a processor, derive your own operator class and pass an instance to the EdgePairs#processed method.

Conceptually, these methods take each edge pair from the edge pair collection and present it to the operator's 'process' method. The result of this call is a list of zero to many output polygons derived from the input edge pair. The output region is the sum over all these individual results.

The magic happens when deep mode edge pair collections are involved. In that case, the processor will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the operator behaves. You need to configure the operator by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using it.

You can skip this step, but the processor algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

For a basic example see the EdgeToPolygonOperator class, with the exception that this incarnation receives edge pairs.

This class has been introduced in version 0.29.

Public constructors

new EdgePairToPolygonOperator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**is_isotropic**		Indicates that the filter has isotropic properties
	void	**is_isotropic_and_scale_invariant**		Indicates that the filter is isotropic and scale invariant
	void	**is_scale_invariant**		Indicates that the filter is scale invariant
[virtual,const]	Polygon[]	**process**	(const EdgePair shape)	Processes a shape
	void	**wants_variants=**	(bool flag)	Sets a value indicating whether the filter prefers cell variants

[const]	bool	wants_variants?	Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void _**manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void _**unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) processors are size or shrink operators. Size or shrink is not dependent on orientation unless size or shrink needs to be different in x and y direction.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) processor is the convex decomposition operator. The decomposition of a polygon into convex parts is an operation that is not depending on scale nor orientation.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) processor is the rotation operator. Rotation is not depending on scale, but on the original orientation as mirrored versions need to be rotated differently.

new

Signature: *[static]* new EdgePairToPolygonOperator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

process

Signature: *[virtual,const]* Polygon[] **process** (const EdgePair shape)

Description: Processes a shape

This method is the actual payload. It needs to be reimplemented in a derived class. If needs to process the input shape and deliver a list of output shapes. The output list may be empty to entirely discard the input shape. It may also contain more than a single shape. In that case, the number of total shapes may grow during application of the processor.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.40. API reference - Class EdgePairToEdgeOperator

Notation used in Ruby API documentation

Module: db

Description: A generic edge-pair-to-edge operator

Edge processors are an efficient way to process edge pairs from an edge pair collection. To apply a processor, derive your own operator class and pass an instance to EdgePairs#processed method.

Conceptually, these methods take each edge from the edge collection and present it to the operator's 'process' method. The result of this call is a list of zero to many output edges derived from the input edge pair. The output edge pair collection is the sum over all these individual results.

The magic happens when deep mode edge pair collections are involved. In that case, the processor will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the operator behaves. You need to configure the operator by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using it.

You can skip this step, but the processor algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

For a basic example see the EdgeToEdgePairOperator class, with the exception that this incarnation has to deliver edges and takes edge pairs.

This class has been introduced in version 0.29.

Public constructors

new EdgePairToEdgeOperator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**is_isotropic**		Indicates that the filter has isotropic properties
	void	**is_isotropic_and_scale_invariant**		Indicates that the filter is isotropic and scale invariant
	void	**is_scale_invariant**		Indicates that the filter is scale invariant
[virtual,const]	Edge[]	**process**	(const EdgePair shape)	Processes a shape
	void	**wants_variants=**	(bool flag)	Sets a value indicating whether the filter prefers cell variants

[const]	bool	**wants_variants?**	Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) processors are size or shrink operators. Size or shrink is not dependent on orientation unless size or shrink needs to be different in x and y direction.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) processor is the convex decomposition operator. The decomposition of a polygon into convex parts is an operation that is not depending on scale nor orientation.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) processor is the rotation operator. Rotation is not depending on scale, but on the original orientation as mirrored versions need to be rotated differently.

new

Signature: *[static]* new EdgePairToEdgeOperator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

process

Signature: *[virtual,const]* Edge[] **process** (const EdgePair shape)

Description: Processes a shape

This method is the actual payload. It needs to be reimplemented in a derived class. If needs to process the input shape and deliver a list of output shapes. The output list may be empty to entirely discard the input shape. It may also contain more than a single shape. In that case, the number of total shapes may grow during application of the processor.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.41. API reference - Class EdgePairs

Notation used in Ruby API documentation

Module: db

Description: EdgePairs (a collection of edge pairs)

Class hierarchy: EdgePairs » ShapeCollection

Edge pairs are used mainly in the context of the DRC functions (width_check, space_check etc.) of Region and Edges. A single edge pair represents two edges participating in a DRC violation. In the two-layer checks (inside, overlap) The first edge represents an edge from the first layer and the second edge an edge from the second layer. For single-layer checks (width, space) the order of the edges is arbitrary.

This class has been introduced in version 0.23.

Public constructors

new EdgePairs ptr	**new**		Default constructor
new EdgePairs ptr	**new**	(EdgePair[] array)	Constructor from an edge pair array
new EdgePairs ptr	**new**	(const EdgePair edge_pair)	Constructor from a single edge pair object
new EdgePairs ptr	**new**	(const Shapes shapes)	Shapes constructor
new EdgePairs ptr	**new**	(const RecursiveShapeIterator shape_iterator)	Constructor from a hierarchical shape set
new EdgePairs ptr	**new**	(const RecursiveShapeIterator shape_iterator, const ICplxTrans trans)	Constructor from a hierarchical shape set with a transformation
new EdgePairs ptr	**new**	(const RecursiveShapeIterator shape_iterator, DeepShapeStore dss)	Creates a hierarchical edge pair collection from an original layer
new EdgePairs ptr	**new**	(const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, const ICplxTrans trans)	Creates a hierarchical edge pair collection from an original layer with a transformation

Public methods

[const]	EdgePairs	**+**	(const EdgePairs other)	Returns the combined edge pair collection of self and the other one
	EdgePairs	**+=**	(const EdgePairs other)	Adds the edge pairs of the other edge pair collection to self
[const]	const EdgePair ptr	**[]**	(unsigned long n)	Returns the nth edge pair
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed

[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const EdgePairs other)	Assigns another object to self
[const]	Box	**bbox**		Return the bounding box of the edge pair collection
	void	**clear**		Clears the edge pair collection
[const]	unsigned long	**count**		Returns the (flat) number of edge pairs in the edge pair collection
[const]	unsigned long	**data_id**		Returns the data ID (a unique identifier for the underlying data storage)
	void	**disable_progress**		Disable progress reporting
[const]	new EdgePairs ptr	**dup**		Creates a copy of self
[const,iter]	EdgePair	**each**		Returns each edge pair of the edge pair collection
[const]	Edges	**edges**		Decomposes the edge pairs into single edges
	void	**enable_progress**	(string label)	Enable progress reporting
	void	**enable_properties**		Enables properties for the given container.
[const]	Region	**extents**		Returns a region with the bounding boxes of the edge pairs
[const]	Region	**extents**	(int d)	Returns a region with the enlarged bounding boxes of the edge pairs
[const]	Region	**extents**	(int dx, int dy)	Returns a region with the enlarged bounding boxes of the edge pairs
	void	**filter**	(const EdgePairFilter ptr filter)	Applies a generic filter in place (replacing the edge pairs from the EdgePair collection)
	void	**filter_properties**	(variant[] keys)	Filters properties by certain keys.
[const]	EdgePairs	**filtered**	(const EdgePairFilter ptr filtered)	Applies a generic filter and returns a filtered copy
[const]	Edges	**first_edges**		Returns the first one of all edges
	void	**flatten**		Explicitly flattens an edge pair collection

[const]	bool	**has_valid_edge_pai**		Returns true if the edge pair collection is flat and individual edge pairs can be accessed randomly
[const]	unsigned long	**hier_count**		Returns the (hierarchical) number of edge pairs in the edge pair collection
	void	**insert**	(const Edge first, const Edge second)	Inserts an edge pair into the collection
	void	**insert**	(const EdgePair edge_pair)	Inserts an edge pair into the collection
	void	**insert**	(const EdgePairs edge_pairs)	Inserts all edge pairs from the other edge pair collection into this edge pair collection
[const]	void	**insert_into**	(Layout ptr layout, unsigned int cell_index, unsigned int layer)	Inserts this edge pairs into the given layout, below the given cell and into the given layer.
[const]	void	**insert_into_as_polyg**	(Layout ptr layout, unsigned int cell_index, unsigned int layer, int e)	Inserts this edge pairs into the given layout, below the given cell and into the given layer.
[const]	bool	**is_deep?**		Returns true if the edge pair collection is a deep (hierarchical) one
[const]	bool	**is_empty?**		Returns true if the collection is empty
[const]	EdgePairs	**join**	(const EdgePairs other)	Returns the combined edge pair collection of self and the other one
	EdgePairs	**join_with**	(const EdgePairs other)	Adds the edge pairs of the other edge pair collection to self
	void	**map_properties**	(map<variant,variant> key_map)	Maps properties by name key.
	EdgePairs	**move**	(const Vector p)	Moves the edge pair collection
	EdgePairs	**move**	(int x, int y)	Moves the edge pair collection
[const]	EdgePairs	**moved**	(const Vector p)	Returns the moved edge pair collection (does not modify self)
[const]	EdgePairs	**moved**	(int x, int y)	Returns the moved edge pair collection (does not modify self)
[const]	Region	**polygons**		Converts the edge pairs to polygons
[const]	Region	**polygons**	(int e)	Converts the edge pairs to polygons
	void	**process**	(const EdgePairOperator ptr process)	Applies a generic edge pair processor in place (replacing the edge pairs from the EdgePairs collection)

[const]	EdgePairs	**processed**	(const EdgePairOperator ptr processed)	Applies a generic edge pair processor and returns a processed copy
[const]	Edges	**processed**	(const EdgePairToEdgeOperator ptr processed)	Applies a generic edge-pair-to-edge processor and returns an edge collection with the results
[const]	Region	**processed**	(const EdgePairToPolygonOperator ptr processed)	Applies a generic edge-pair-to-polygon processor and returns an Region with the results
	void	**remove_properties**		Removes properties for the given container.
[const]	Edges	**second_edges**		Returns the second one of all edges
	void	**swap**	(EdgePairs other)	Swap the contents of this collection with the contents of another collection
[const]	string	**to_s**		Converts the edge pair collection to a string
[const]	string	**to_s**	(unsigned long max_count)	Converts the edge pair collection to a string
	EdgePairs	**transform**	(const Trans t)	Transform the edge pair collection (modifies self)
	EdgePairs	**transform**	(const ICplxTrans t)	Transform the edge pair collection with a complex transformation (modifies self)
	EdgePairs	**transform**	(const IMatrix2d t)	Transform the edge pair collection (modifies self)
	EdgePairs	**transform**	(const IMatrix3d t)	Transform the edge pair collection (modifies self)
[const]	EdgePairs	**transformed**	(const Trans t)	Transform the edge pair collection
[const]	EdgePairs	**transformed**	(const ICplxTrans t)	Transform the edge pair collection with a complex transformation
[const]	EdgePairs	**transformed**	(const IMatrix2d t)	Transform the edge pair collection
[const]	EdgePairs	**transformed**	(const IMatrix3d t)	Transform the edge pair collection
[const]	EdgePairs	**with_angle**	(double angle, bool inverse)	Filter the edge pairs by orientation of their edges
[const]	EdgePairs	**with_angle**	(double min_angle, double max_angle, bool inverse, bool include_min_angle = true, bool include_max_angle = false)	Filter the edge pairs by orientation of their edges

[const]	EdgePairs	**with_angle**	(Edges::EdgeType type, bool inverse)	Filter the edge pairs by orientation of their edges
[const]	EdgePairs	**with_angle_both**	(double angle, bool inverse)	Filter the edge pairs by orientation of both of their edges
[const]	EdgePairs	**with_angle_both**	(double min_angle, double max_angle, bool inverse, bool include_min_angle = true, bool include_max_angle = false)	Filter the edge pairs by orientation of both of their edges
[const]	EdgePairs	**with_angle_both**	(Edges::EdgeType type, bool inverse)	Filter the edge pairs by orientation of their edges
[const]	EdgePairs	**with_area**	(long area, bool inverse)	Filters the edge pairs by the enclosed area
[const]	EdgePairs	**with_area**	(long min_area, long max_area, bool inverse)	Filters the edge pairs by the enclosed area
[const]	EdgePairs	**with_distance**	(unsigned int distance, bool inverse)	Filters the edge pairs by the distance of the edges
[const]	EdgePairs	**with_distance**	(variant min_distance, variant max_distance, bool inverse)	Filters the edge pairs by the distance of the edges
[const]	EdgePairs	**with_internal_angle**	(double angle, bool inverse)	Filters the edge pairs by the angle between their edges
[const]	EdgePairs	**with_internal_angle**	(double min_angle, double max_angle, bool inverse, bool include_min_angle = true, bool include_max_angle = false)	Filters the edge pairs by the angle between their edges
[const]	EdgePairs	**with_length**	(unsigned int length, bool inverse)	Filters the edge pairs by length of one of their edges
[const]	EdgePairs	**with_length**	(variant min_length, variant max_length, bool inverse)	Filters the edge pairs by length of one of their edges
[const]	EdgePairs	**with_length_both**	(unsigned int length, bool inverse)	Filters the edge pairs by length of both of their edges

[const]	EdgePairs	**with_length_both**	(variant min_length, variant max_length, bool inverse)	Filters the edge pairs by length of both of their edges
[const]	void	**write**	(string filename)	Writes the region to a file

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	unsigned long	**size**		Use of this method is deprecated. Use count instead
	EdgePairs	**transform_icplx**	(const ICplxTrans t)	Use of this method is deprecated. Use transform instead
[const]	EdgePairs	**transformed_icplx**	(const ICplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

+

Signature: *[const]* EdgePairs **+** (const EdgePairs other)

Description: Returns the combined edge pair collection of self and the other one

Returns: The resulting edge pair collection

This operator adds the edge pairs of the other collection to self and returns a new combined set.

This method has been introduced in version 0.24. The 'join' alias has been introduced in version 0.28.12.

+=

Signature: EdgePairs **+=** (const EdgePairs other)

Description: Adds the edge pairs of the other edge pair collection to self

Returns: The edge pair collection after modification (self)

This operator adds the edge pairs of the other collection to self.

This method has been introduced in version 0.24.

Note that in Ruby, the '+=' operator actually does not exist, but is emulated by '+' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'join_with' instead.

The 'join_with' alias has been introduced in version 0.28.12.

[]

Signature: *[const]* const EdgePair ptr **[]** (unsigned long n)

Description: Returns the nth edge pair

This method returns nil if the index is out of range. It is available for flat edge pairs only - i.e. those for which has_valid_edge_pairs? is true. Use flatten to explicitly flatten an edge pair collection.

The each iterator is the more general approach to access the edge pairs.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const EdgePairs other)

Description: Assigns another object to self

bbox

Signature: *[const]* Box **bbox**

Description: Return the bounding box of the edge pair collection

The bounding box is the box enclosing all points of all edge pairs.

clear

Signature: void **clear**

Description: Clears the edge pair collection

count

Signature: *[const]* unsigned long **count**

Description: Returns the (flat) number of edge pairs in the edge pair collection

The count is computed 'as if flat', i.e. edge pairs inside a cell are multiplied by the number of times a cell is instantiated.

Starting with version 0.27, the method is called 'count' for consistency with Region. 'size' is still provided as an alias.

Python specific notes:
This method is also available as 'len(object)'.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

data_id

Signature: *[const]* unsigned long **data_id**

Description: Returns the data ID (a unique identifier for the underlying data storage)

This method has been added in version 0.26.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disable_progress

Signature: void **disable_progress**

Description: Disable progress reporting

Calling this method will disable progress reporting. See enable_progress.

dup

Signature: *[const]* new EdgePairs ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each

Signature: *[const,iter]* EdgePair **each**

Description: Returns each edge pair of the edge pair collection

Python specific notes:
This method enables iteration of the object.

edges

Signature: *[const]* Edges **edges**

Description: Decomposes the edge pairs into single edges

Returns: An edge collection containing the individual edges

enable_progress

Signature: void **enable_progress** (string label)

Description: Enable progress reporting

After calling this method, the edge pair collection will report the progress through a progress bar while expensive operations are running. The label is a text which is put in front of the progress bar. Using a progress bar will imply a performance penalty of a few percent typically.

enable_properties

Signature: void **enable_properties**

Description: Enables properties for the given container.

This method has an effect mainly on original layers and will import properties from such layers. By default, properties are not enabled on original layers. Alternatively you can apply filter_properties or map_properties to enable properties with a specific name key.

This method has been introduced in version 0.28.4.

extents

(1) Signature: *[const]* Region **extents**

Description: Returns a region with the bounding boxes of the edge pairs

This method will return a region consisting of the bounding boxes of the edge pairs. The boxes will not be merged, so it is possible to determine overlaps of these boxes for example.

(2) Signature: *[const]* Region **extents** (int d)

Description: Returns a region with the enlarged bounding boxes of the edge pairs

This method will return a region consisting of the bounding boxes of the edge pairs enlarged by the given distance d. The enlargement is specified per edge, i.e the width and height will be increased by 2*d. The boxes will not be merged, so it is possible to determine overlaps of these boxes for example.

(3) Signature: *[const]* Region **extents** (int dx, int dy)

Description: Returns a region with the enlarged bounding boxes of the edge pairs

This method will return a region consisting of the bounding boxes of the edge pairs enlarged by the given distance dx in x direction and dy in y direction. The enlargement is specified per edge, i.e the width will be increased by 2*dx. The boxes will not be merged, so it is possible to determine overlaps of these boxes for example.

filter

Signature: void **filter** (const EdgePairFilter ptr filter)

Description: Applies a generic filter in place (replacing the edge pairs from the EdgePair collection)

See EdgePairFilter for a description of this feature.

This method has been introduced in version 0.29.

filter_properties

Signature: void **filter_properties** (variant[] keys)

Description: Filters properties by certain keys.

Calling this method on a container will reduce the properties to values with name keys from the 'keys' list. As a side effect, this method enables properties on original layers.

This method has been introduced in version 0.28.4.

filtered

Signature: *[const]* EdgePairs **filtered** (const EdgePairFilter ptr filtered)

Description: Applies a generic filter and returns a filtered copy

See EdgePairFilter for a description of this feature.

This method has been introduced in version 0.29.

first_edges

Signature: *[const]* Edges **first_edges**

Description: Returns the first one of all edges

Returns: An edge collection containing the first edges

flatten

Signature: void **flatten**

Description: Explicitly flattens an edge pair collection

If the collection is already flat (i.e. has_valid_edge_pairs? returns true), this method will not change the collection.

This method has been introduced in version 0.26.

has_valid_edge_pairs?

Signature: *[const]* bool **has_valid_edge_pairs?**

Description: Returns true if the edge pair collection is flat and individual edge pairs can be accessed randomly

This method has been introduced in version 0.26.

hier_count

Signature: *[const]* unsigned long **hier_count**

Description: Returns the (hierarchical) number of edge pairs in the edge pair collection

The count is computed 'hierarchical', i.e. edge pairs inside a cell are counted once even if the cell is instantiated multiple times.

This method has been introduced in version 0.27.

insert

(1) Signature: void **insert** (const Edge first, const Edge second)

Description: Inserts an edge pair into the collection

(2) Signature: void **insert** (const EdgePair edge_pair)

Description: Inserts an edge pair into the collection

(3) Signature: void **insert** (const EdgePairs edge_pairs)

Description: Inserts all edge pairs from the other edge pair collection into this edge pair collection

This method has been introduced in version 0.25.

insert_into

Signature: *[const]* void **insert_into** (Layout ptr layout, unsigned int cell_index, unsigned int layer)

Description: Inserts this edge pairs into the given layout, below the given cell and into the given layer.

If the edge pair collection is a hierarchical one, a suitable hierarchy will be built below the top cell or and existing hierarchy will be reused.

This method has been introduced in version 0.26.

insert_into_as_polygons

Signature: *[const]* void **insert_into_as_polygons** (Layout ptr layout, unsigned int cell_index, unsigned int layer, int e)

Description: Inserts this edge pairs into the given layout, below the given cell and into the given layer.

If the edge pair collection is a hierarchical one, a suitable hierarchy will be built below the top cell or and existing hierarchy will be reused.

The edge pairs will be converted to polygons with the enlargement value given be 'e'.

This method has been introduced in version 0.26.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_deep?

Signature: *[const]* bool **is_deep?**

Description: Returns true if the edge pair collection is a deep (hierarchical) one

This method has been added in version 0.26.

is_empty?

Signature: *[const]* bool **is_empty?**

Description: Returns true if the collection is empty

join

Signature: *[const]* EdgePairs **join** (const EdgePairs other)

Description: Returns the combined edge pair collection of self and the other one

Returns: The resulting edge pair collection

This operator adds the edge pairs of the other collection to self and returns a new combined set.

This method has been introduced in version 0.24. The 'join' alias has been introduced in version 0.28.12.

join_with

Signature: EdgePairs **join_with** (const EdgePairs other)

Description: Adds the edge pairs of the other edge pair collection to self

Returns: The edge pair collection after modification (self)

This operator adds the edge pairs of the other collection to self.

This method has been introduced in version 0.24.

Note that in Ruby, the '+=' operator actually does not exist, but is emulated by '+' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'join_with' instead.

The 'join_with' alias has been introduced in version 0.28.12.

map_properties

Signature: void **map_properties** (map<variant,variant> key_map)

Description: Maps properties by name key.

Calling this method on a container will reduce the properties to values with name keys from the 'keys' hash and renames the properties. Properties not listed in the key map will be removed. As a side effect, this method enables properties on original layers.

This method has been introduced in version 0.28.4.

move

(1) Signature: EdgePairs **move** (const Vector p)

Description: Moves the edge pair collection

p:	The distance to move the edge pairs.
Returns:	The moved edge pairs (self).

Moves the edge pairs by the given offset and returns the moved edge pair collection. The edge pair collection is overwritten.

Starting with version 0.25 the displacement is of vector type.

(2) Signature: EdgePairs **move** (int x, int y)

Description: Moves the edge pair collection

x:	The x distance to move the edge pairs.
y:	The y distance to move the edge pairs.
Returns:	The moved edge pairs (self).

Moves the edge pairs by the given offset and returns the moved edge pairs. The edge pair collection is overwritten.

moved

(1) Signature: *[const]* EdgePairs **moved** (const Vector p)

Description: Returns the moved edge pair collection (does not modify self)

p:	The distance to move the edge pairs.
Returns:	The moved edge pairs.

Moves the edge pairs by the given offset and returns the moved edge pairs. The edge pair collection is not modified.

Starting with version 0.25 the displacement is of vector type.

(2) Signature: *[const]* EdgePairs **moved** (int x, int y)

Description: Returns the moved edge pair collection (does not modify self)

x:	The x distance to move the edge pairs.
y:	The y distance to move the edge pairs.
Returns:	The moved edge pairs.

Moves the edge pairs by the given offset and returns the moved edge pairs. The edge pair collection is not modified.

new

(1) Signature: *[static]* new EdgePairs ptr **new**

Description: Default constructor

This constructor creates an empty edge pair collection.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new EdgePairs ptr **new** (EdgePair[] array)

Description: Constructor from an edge pair array

This constructor creates an edge pair collection from an array of EdgePair objects.

This constructor has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new EdgePairs ptr **new** (const EdgePair edge_pair)

Description: Constructor from a single edge pair object

This constructor creates an edge pair collection with a single edge pair.

This constructor has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new EdgePairs ptr **new** (const Shapes shapes)

Description: Shapes constructor

This constructor creates an edge pair collection from a Shapes collection.

This constructor has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new EdgePairs ptr **new** (const RecursiveShapeIterator shape_iterator)

Description: Constructor from a hierarchical shape set

This constructor creates an edge pair collection from the shapes delivered by the given recursive shape iterator. Only edge pairs are taken from the shape set and other shapes are ignored. This method allows feeding the edge pair collection from a hierarchy of cells. Edge pairs in layout objects are somewhat special as most formats don't support reading or writing of edge pairs. Still they are useful objects and can be created and manipulated inside layouts.

```
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
r = RBA::EdgePairs::new(layout.begin_shapes(cell, layer))
```

This constructor has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new EdgePairs ptr **new** (const RecursiveShapeIterator shape_iterator, const ICplxTrans trans)

Description: Constructor from a hierarchical shape set with a transformation

This constructor creates an edge pair collection from the shapes delivered by the given recursive shape iterator. Only edge pairs are taken from the shape set and other shapes are ignored. The given transformation is applied to each edge pair taken. This method allows feeding the edge pair collection from a hierarchy of cells. The transformation is useful to scale to a specific database unit for example. Edge pairs in layout objects are somewhat special as most formats don't support reading or writing of edge pairs. Still they are useful objects and can be created and manipulated inside layouts.

```
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
dbu    = 0.1 # the target database unit
r = RBA::EdgePairs::new(layout.begin_shapes(cell, layer),
 RBA::ICplxTrans::new(layout.dbu / dbu))
```

This constructor has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new EdgePairs ptr **new** (const RecursiveShapeIterator shape_iterator, DeepShapeStore dss)

Description: Creates a hierarchical edge pair collection from an original layer

This constructor creates an edge pair collection from the shapes delivered by the given recursive shape iterator. This version will create a hierarchical edge pair collection which supports hierarchical operations. Edge pairs in layout objects are somewhat special as most formats don't support reading or writing of edge pairs. Still they are useful objects and can be created and manipulated inside layouts.

```
dss    = RBA::DeepShapeStore::new
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
r = RBA::EdgePairs::new(layout.begin_shapes(cell, layer))
```

This constructor has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new EdgePairs ptr **new** (const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, const ICplxTrans trans)

Description: Creates a hierarchical edge pair collection from an original layer with a transformation

This constructor creates an edge pair collection from the shapes delivered by the given recursive shape iterator. This version will create a hierarchical edge pair collection which supports hierarchical operations. The transformation is useful to scale to a specific database unit for example. Edge pairs in layout objects are somewhat special as most formats don't support reading or writing of edge pairs. Still they are useful objects and can be created and manipulated inside layouts.

```
dss    = RBA::DeepShapeStore::new
```

```
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
dbu    = 0.1 # the target database unit
r = RBA::EdgePairs::new(layout.begin_shapes(cell, layer),
 RBA::ICplxTrans::new(layout.dbu / dbu))
```

This constructor has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

polygons

(1) Signature: *[const]* Region **polygons**

Description: Converts the edge pairs to polygons

This method creates polygons from the edge pairs. Each polygon will be a triangle or quadrangle which connects the start and end points of the edges forming the edge pair.

(2) Signature: *[const]* Region **polygons** (int e)

Description: Converts the edge pairs to polygons

This method creates polygons from the edge pairs. Each polygon will be a triangle or quadrangle which connects the start and end points of the edges forming the edge pair. This version allows one to specify an enlargement which is applied to the edges. The length of the edges is modified by applying the enlargement and the edges are shifted by the enlargement. By specifying an enlargement it is possible to give edge pairs an area which otherwise would not have one (coincident edges, two point-like edges).

process

Signature: void **process** (const EdgePairOperator ptr process)

Description: Applies a generic edge pair processor in place (replacing the edge pairs from the EdgePairs collection)

See EdgePairProcessor for a description of this feature.

This method has been introduced in version 0.29.

processed

(1) Signature: *[const]* EdgePairs **processed** (const EdgePairOperator ptr processed)

Description: Applies a generic edge pair processor and returns a processed copy

See EdgePairProcessor for a description of this feature.

This method has been introduced in version 0.29.

(2) Signature: *[const]* Edges **processed** (const EdgePairToEdgeOperator ptr processed)

Description: Applies a generic edge-pair-to-edge processor and returns an edge collection with the results

See EdgePairToEdgeProcessor for a description of this feature.

This method has been introduced in version 0.29.

(3) Signature: *[const]* Region **processed** (const EdgePairToPolygonOperator ptr processed)

Description: Applies a generic edge-pair-to-polygon processor and returns an Region with the results

See EdgePairToPolygonProcessor for a description of this feature.

This method has been introduced in version 0.29.

remove_properties

Signature: void **remove_properties**

Description: Removes properties for the given container.

This will remove all properties on the given container.

This method has been introduced in version 0.28.4.

second_edges

Signature: *[const]* Edges **second_edges**

Description: Returns the second one of all edges

Returns: An edge collection containing the second edges

size

Signature: *[const]* unsigned long **size**

Description: Returns the (flat) number of edge pairs in the edge pair collection

Use of this method is deprecated. Use count instead

The count is computed 'as if flat', i.e. edge pairs inside a cell are multiplied by the number of times a cell is instantiated.

Starting with version 0.27, the method is called 'count' for consistency with Region. 'size' is still provided as an alias.

Python specific notes:
This method is also available as 'len(object)'.

swap

Signature: void **swap** (EdgePairs other)

Description: Swap the contents of this collection with the contents of another collection

This method is useful to avoid excessive memory allocation in some cases. For managed memory languages such as Ruby, those cases will be rare.

to_s

(1) Signature: *[const]* string **to_s**

Description: Converts the edge pair collection to a string

The length of the output is limited to 20 edge pairs to avoid giant strings on large regions. For full output use "to_s" with a maximum count parameter.

Python specific notes:
This method is also available as 'str(object)'.

(2) Signature: *[const]* string **to_s** (unsigned long max_count)

Description: Converts the edge pair collection to a string

This version allows specification of the maximum number of edge pairs contained in the string.

transform

(1) Signature: EdgePairs **transform** (const Trans t)

Description: Transform the edge pair collection (modifies self)

t: The transformation to apply.

Returns: The transformed edge pair collection.

Transforms the edge pair collection with the given transformation. This version modifies the edge pair collection and returns a reference to self.

(2) Signature: EdgePairs **transform** (const ICplxTrans t)

Description: Transform the edge pair collection with a complex transformation (modifies self)

t: The transformation to apply.

Returns: The transformed edge pair collection.

Transforms the edge pair collection with the given transformation. This version modifies the edge pair collection and returns a reference to self.

(3) Signature: EdgePairs **transform** (const IMatrix2d t)

Description: Transform the edge pair collection (modifies self)

t: The transformation to apply.

Returns: The transformed edge pair collection.

Transforms the edge pair collection with the given 2d matrix transformation. This version modifies the edge pair collection and returns a reference to self.

This variant has been introduced in version 0.27.

(4) Signature: EdgePairs **transform** (const IMatrix3d t)

Description: Transform the edge pair collection (modifies self)

t: The transformation to apply.

Returns: The transformed edge pair collection.

Transforms the edge pair collection with the given 3d matrix transformation. This version modifies the edge pair collection and returns a reference to self.

This variant has been introduced in version 0.27.

transform_icplx

Signature: EdgePairs **transform_icplx** (const ICplxTrans t)

Description: Transform the edge pair collection with a complex transformation (modifies self)

t: The transformation to apply.

Returns: The transformed edge pair collection.

Use of this method is deprecated. Use transform instead

Transforms the edge pair collection with the given transformation. This version modifies the edge pair collection and returns a reference to self.

transformed

(1) Signature: *[const]* EdgePairs **transformed** (const Trans t)

Description: Transform the edge pair collection

t: The transformation to apply.

Returns: The transformed edge pairs.

Transforms the edge pairs with the given transformation. Does not modify the edge pair collection but returns the transformed edge pairs.

(2) Signature: *[const]* EdgePairs **transformed** (const ICplxTrans t)

Description: Transform the edge pair collection with a complex transformation

t: The transformation to apply.

Returns: The transformed edge pairs.

Transforms the edge pairs with the given complex transformation. Does not modify the edge pair collection but returns the transformed edge pairs.

(3) Signature: *[const]* EdgePairs **transformed** (const IMatrix2d t)

Description: Transform the edge pair collection

t: The transformation to apply.

Returns: The transformed edge pairs.

Transforms the edge pairs with the given 2d matrix transformation. Does not modify the edge pair collection but returns the transformed edge pairs.

This variant has been introduced in version 0.27.

(4) Signature: *[const]* EdgePairs **transformed** (const IMatrix3d t)

Description: Transform the edge pair collection

t: The transformation to apply.

Returns: The transformed edge pairs.

Transforms the edge pairs with the given 3d matrix transformation. Does not modify the edge pair collection but returns the transformed edge pairs.

This variant has been introduced in version 0.27.

transformed_icplx

Signature: *[const]* EdgePairs **transformed_icplx** (const ICplxTrans t)

Description: Transform the edge pair collection with a complex transformation

t: The transformation to apply.

Returns: The transformed edge pairs.

Use of this method is deprecated. Use transformed instead

Transforms the edge pairs with the given complex transformation. Does not modify the edge pair collection but returns the transformed edge pairs.

with_angle

(1) Signature: *[const]* EdgePairs **with_angle** (double angle, bool inverse)

Description: Filter the edge pairs by orientation of their edges

Filters the edge pairs in the edge pair collection by orientation. If "inverse" is false, only edge pairs with at least one edge having the given angle to the x-axis are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

This will filter edge pairs with at least one horizontal edge:

```
horizontal = edge_pairs.with_angle(0, false)
```

Note that the inverse **result** of with_angle is delivered by with_angle_both with the inverse flag set as edge pairs are unselected when both edges fail to meet the criterion. I.e

```
result = edge_pairs.with_angle(0, false)
others = edge_pairs.with_angle_both(0, true)
```

This method has been added in version 0.27.1.

(2) Signature: *[const]* EdgePairs **with_angle** (double min_angle, double max_angle, bool inverse, bool include_min_angle = true, bool include_max_angle = false)

Description: Filter the edge pairs by orientation of their edges

Filters the edge pairs in the edge pair collection by orientation. If "inverse" is false, only edge pairs with at least one edge having an angle between min_angle and max_angle are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

With "include_min_angle" set to true (the default), the minimum angle is included in the criterion while with false, the minimum angle itself is not included. Same for "include_max_angle" where the default is false, meaning the maximum angle is not included in the range.

Note that the inverse **result** of with_angle is delivered by with_angle_both with the inverse flag set as edge pairs are unselected when both edges fail to meet the criterion. I.e

```
result = edge_pairs.with_angle(0, 45, false)
others = edge_pairs.with_angle_both(0, 45, true)
```

This method has been added in version 0.27.1.

(3) Signature: *[const]* EdgePairs **with_angle** (Edges::EdgeType type, bool inverse)

Description: Filter the edge pairs by orientation of their edges

Filters the edge pairs in the edge pair collection by orientation. If "inverse" is false, only edge pairs with at least one edge having an angle of the given type are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

This version allows specifying an edge type instead of an angle. Edge types include multiple distinct orientations and are specified using one of the Edges#OrthoEdges, Edges#DiagonalEdges or Edges#OrthoDiagonalEdges types.

Note that the inverse **result** of with_angle is delivered by with_angle_both with the inverse flag set as edge pairs are unselected when both edges fail to meet the criterion. I.e

```
result = edge_pairs.with_angle(RBA::Edges::Ortho, false)
others = edge_pairs.with_angle_both(RBA::Edges::Ortho, true)
```

This method has been added in version 0.28.

with_angle_both

(1) Signature: *[const]* EdgePairs **with_angle_both** (double angle, bool inverse)

Description: Filter the edge pairs by orientation of both of their edges

Filters the edge pairs in the edge pair collection by orientation. If "inverse" is false, only edge pairs with both edges having the given angle to the x-axis are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

This will filter edge pairs with at least one horizontal edge:

```
horizontal = edge_pairs.with_angle_both(0, false)
```

Note that the inverse **result** of with_angle_both is delivered by with_angle with the inverse flag set as edge pairs are unselected when one edge fails to meet the criterion. I.e

```
result = edge_pairs.with_angle_both(0, false)
others = edge_pairs.with_angle(0, true)
```

This method has been added in version 0.27.1.

(2) Signature: *[const]* EdgePairs **with_angle_both** (double min_angle, double max_angle, bool inverse, bool include_min_angle = true, bool include_max_angle = false)

Description: Filter the edge pairs by orientation of both of their edges

Filters the edge pairs in the edge pair collection by orientation. If "inverse" is false, only edge pairs with both edges having an angle between min_angle and max_angle are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

With "include_min_angle" set to true (the default), the minimum angle is included in the criterion while with false, the minimum angle itself is not included. Same for "include_max_angle" where the default is false, meaning the maximum angle is not included in the range.

Note that the inverse **result** of with_angle_both is delivered by with_angle with the inverse flag set as edge pairs are unselected when one edge fails to meet the criterion. I.e

```
result = edge_pairs.with_angle_both(0, 45, false)
others = edge_pairs.with_angle(0, 45, true)
```

This method has been added in version 0.27.1.

(3) Signature: *[const]* EdgePairs **with_angle_both** (Edges::EdgeType type, bool inverse)

Description: Filter the edge pairs by orientation of their edges

Filters the edge pairs in the edge pair collection by orientation. If "inverse" is false, only edge pairs with both edges having an angle of the given type are returned. If "inverse" is true, edge pairs not fulfilling this criterion for both edges are returned.

This version allows specifying an edge type instead of an angle. Edge types include multiple distinct orientations and are specified using one of the Edges#OrthoEdges, Edges#DiagonalEdges or Edges#OrthoDiagonalEdges types.

Note that the inverse **result** of with_angle_both is delivered by with_angle with the inverse flag set as edge pairs are unselected when one edge fails to meet the criterion. I.e

```
result = edge_pairs.with_angle_both(RBA::Edges::Ortho, false)
others = edge_pairs.with_angle(RBA::Edges::Ortho, true)
```

This method has been added in version 0.28.

with_area

(1) Signature: *[const]* EdgePairs **with_area** (long area, bool inverse)

Description: Filters the edge pairs by the enclosed area

Filters the edge pairs in the edge pair collection by enclosed area. If "inverse" is false, only edge pairs with the given area are returned. If "inverse" is true, edge pairs not with the given area are returned.

This method has been added in version 0.27.2.

(2) Signature: *[const]* EdgePairs **with_area** (long min_area, long max_area, bool inverse)

Description: Filters the edge pairs by the enclosed area

Filters the edge pairs in the edge pair collection by enclosed area. If "inverse" is false, only edge pairs with an area between min_area and max_area (max_area itself is excluded) are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

This method has been added in version 0.27.2.

with_distance

(1) Signature: *[const]* EdgePairs **with_distance** (unsigned int distance, bool inverse)

Description: Filters the edge pairs by the distance of the edges

Filters the edge pairs in the edge pair collection by distance of the edges. If "inverse" is false, only edge pairs where both edges have the given distance are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

Distance is measured as the shortest distance between any of the points on the edges.

This method has been added in version 0.27.1.

(2) Signature: *[const]* EdgePairs **with_distance** (variant min_distance, variant max_distance, bool inverse)

Description: Filters the edge pairs by the distance of the edges

Filters the edge pairs in the edge pair collection by distance of the edges. If "inverse" is false, only edge pairs where both edges have a distance between min_distance and max_distance (max_distance itself is excluded) are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

Distance is measured as the shortest distance between any of the points on the edges.

This method has been added in version 0.27.1.

with_internal_angle

(1) Signature: *[const]* EdgePairs **with_internal_angle** (double angle, bool inverse)

Description: Filters the edge pairs by the angle between their edges

Filters the edge pairs in the edge pair collection by the angle between their edges. If "inverse" is false, only edge pairs with the given angle are returned. If "inverse" is true, edge pairs not with the given angle are returned.

The angle is measured between the two edges. It is between 0 (parallel or anti-parallel edges) and 90 degree (perpendicular edges).

This method has been added in version 0.27.2.

(2) Signature: *[const]* EdgePairs **with_internal_angle** (double min_angle, double max_angle, bool inverse, bool include_min_angle = true, bool include_max_angle = false)

Description: Filters the edge pairs by the angle between their edges

Filters the edge pairs in the edge pair collection by the angle between their edges. If "inverse" is false, only edge pairs with an angle between min_angle and max_angle (max_angle itself is excluded) are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

The angle is measured between the two edges. It is between 0 (parallel or anti-parallel edges) and 90 degree (perpendicular edges).

With "include_min_angle" set to true (the default), the minimum angle is included in the criterion while with false, the minimum angle itself is not included. Same for "include_max_angle" where the default is false, meaning the maximum angle is not included in the range.

This method has been added in version 0.27.2.

with_length

(1) Signature: *[const]* EdgePairs **with_length** (unsigned int length, bool inverse)

Description: Filters the edge pairs by length of one of their edges

Filters the edge pairs in the edge pair collection by length of at least one of their edges. If "inverse" is false, only edge pairs with at least one edge having the given length are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

This method has been added in version 0.27.1.

(2) Signature: *[const]* EdgePairs **with_length** (variant min_length, variant max_length, bool inverse)

Description: Filters the edge pairs by length of one of their edges

Filters the edge pairs in the edge pair collection by length of at least one of their edges. If "inverse" is false, only edge pairs with at least one edge having a length between min_length and max_length (excluding max_length itself) are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

If you don't want to specify a lower or upper limit, pass nil to that parameter.

This method has been added in version 0.27.1.

with_length_both

(1) Signature: *[const]* EdgePairs **with_length_both** (unsigned int length, bool inverse)

Description: Filters the edge pairs by length of both of their edges

Filters the edge pairs in the edge pair collection by length of both of their edges. If "inverse" is false, only edge pairs where both edges have the given length are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

This method has been added in version 0.27.1.

(2) Signature: *[const]* EdgePairs **with_length_both** (variant min_length, variant max_length, bool inverse)

Description: Filters the edge pairs by length of both of their edges

Filters the edge pairs in the edge pair collection by length of both of their edges. If "inverse" is false, only edge pairs with both edges having a length between min_length and max_length (excluding max_length itself) are returned. If "inverse" is true, edge pairs not fulfilling this criterion are returned.

If you don't want to specify a lower or upper limit, pass nil to that parameter.

This method has been added in version 0.27.1.

write

Signature: *[const]* void **write** (string filename)

Description: Writes the region to a file

This method is provided for debugging purposes. It writes the object to a flat layer 0/0 in a single top cell.

This method has been introduced in version 0.29.

4.42. API reference - Class EdgeProcessor

Notation used in Ruby API documentation

Module: db

Description: The edge processor (boolean, sizing, merge)

The edge processor implements the boolean and edge set operations (size, merge). Because the edge processor might allocate resources which can be reused in later operations, it is implemented as an object that can be used several times.

Here is a simple example of how to use the edge processor:

```
ep = RBA::EdgeProcessor::new
# Prepare two boxes
a = [ RBA::Polygon::new(RBA::Box::new(0, 0, 300, 300)) ]
b = [ RBA::Polygon::new(RBA::Box::new(100, 100, 200, 200)) ]
# Run an XOR -> creates a polygon with a hole, since the 'resolve_holes' parameter
# is false:
out = ep.boolean_p2p(a, b, RBA::EdgeProcessor::ModeXor, false, false)
out.to_s    # -> [(0,0;0,300;300,300;300,0/100,100;200,100;200,200;100,200)]
```

Public constructors

new EdgeProcessor ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const EdgeProcessor other)	Assigns another object to self
	Edge[]	**boolean_e2e**	(Edge[] a, Edge[] b, int mode)	Boolean operation for a set of given edges, creating edges
	Polygon[]	**boolean_e2p**	(Edge[] a, Edge[] b, int mode, bool resolve_holes, bool min_coherence)	Boolean operation for a set of given edges, creating polygons

	Edge[]	**boolean_p2e**	(Polygon[] a, Polygon[] b, int mode)	Boolean operation for a set of given polygons, creating edges
	Polygon[]	**boolean_p2p**	(Polygon[] a, Polygon[] b, int mode, bool resolve_holes, bool min_coherence)	Boolean operation for a set of given polygons, creating polygons
	void	**disable_progress**		Disable progress reporting
[const]	new EdgeProcessor ptr	**dup**		Creates a copy of self
	void	**enable_progress**	(string label)	Enable progress reporting
	Edge[]	**merge_p2e**	(Polygon[] in, unsigned int min_wc)	Merge the given polygons
	Polygon[]	**merge_p2p**	(Polygon[] in, unsigned int min_wc, bool resolve_holes, bool min_coherence)	Merge the given polygons
	Edge[]	**simple_merge_e2e**	(Edge[] in)	Merge the given edges in a simple "non-zero wrapcount" fashion
	Edge[]	**simple_merge_e2e**	(Edge[] in, int mode)	Merge the given polygons and specify the merge mode
	Polygon[]	**simple_merge_e2p**	(Edge[] in, bool resolve_holes, bool min_coherence)	Merge the given edges in a simple "non-zero wrapcount" fashion into polygons
	Polygon[]	**simple_merge_e2p**	(Edge[] in, bool resolve_holes, bool min_coherence, int mode)	Merge the given polygons and specify the merge mode
	Edge[]	**simple_merge_p2e**	(Polygon[] in)	Merge the given polygons in a simple "non-zero wrapcount" fashion
	Edge[]	**simple_merge_p2e**	(Polygon[] in, int mode)	Merge the given polygons and specify the merge mode
	Polygon[]	**simple_merge_p2p**	(Polygon[] in, bool resolve_holes, bool min_coherence)	Merge the given polygons in a simple "non-zero wrapcount" fashion into polygons
	Polygon[]	**simple_merge_p2p**	(Polygon[] in, bool resolve_holes, bool min_coherence, int mode)	Merge the given polygons and specify the merge mode
	Edge[]	**size_p2e**	(Polygon[] in, int dx, int dy,	Size the given polygons

		unsigned int mode)	
Edge[]	size_p2e	(Polygon[] in, int d, unsigned int mode)	Size the given polygons (isotropic)
Polygon[]	size_p2p	(Polygon[] in, int dx, int dy, unsigned int mode, bool resolve_holes, bool min_coherence)	Size the given polygons into polygons
Polygon[]	size_p2p	(Polygon[] in, int d, unsigned int mode, bool resolve_holes, bool min_coherence)	Size the given polygons into polygons (isotropic)

Public static methods and constants

int	ModeANotB	boolean method's mode value for A NOT B operation
int	ModeAnd	boolean method's mode value for AND operation
int	ModeBNotA	boolean method's mode value for B NOT A operation
int	ModeOr	boolean method's mode value for OR operation
int	ModeXor	boolean method's mode value for XOR operation

Deprecated methods (protected, public, static, non-static and constructors)

Edge[]	boolean	(Polygon[] a, Polygon[] b, int mode)	Use of this method is deprecated. Use boolean_p2e instead
Edge[]	boolean	(Edge[] a, Edge[] b, int mode)	Use of this method is deprecated. Use boolean_e2e instead
Polygon[]	boolean_to_polygon	(Polygon[] a, Polygon[] b, int mode, bool resolve_holes, bool min_coherence)	Use of this method is deprecated. Use boolean_p2p instead
Polygon[]	boolean_to_polygon	(Edge[] a, Edge[] b, int mode, bool resolve_holes, bool min_coherence)	Use of this method is deprecated. Use boolean_e2p instead
void	create		Use of this method is deprecated. Use _create instead

	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
	Edge[]	**merge**	(Polygon[] in, unsigned int min_wc)	Use of this method is deprecated. Use merge_p2e instead
	Polygon[]	**merge_to_polygon**	(Polygon[] in, unsigned int min_wc, bool resolve_holes, bool min_coherence)	Use of this method is deprecated. Use merge_p2p instead
[static]	int	**mode_and**		Use of this method is deprecated. Use ModeAnd instead
[static]	int	**mode_anotb**		Use of this method is deprecated. Use ModeANotB instead
[static]	int	**mode_bnota**		Use of this method is deprecated. Use ModeBNotA instead
[static]	int	**mode_or**		Use of this method is deprecated. Use ModeOr instead
[static]	int	**mode_xor**		Use of this method is deprecated. Use ModeXor instead
	Edge[]	**simple_merge**	(Polygon[] in)	Use of this method is deprecated. Use simple_merge_p2e instead
	Edge[]	**simple_merge**	(Polygon[] in, int mode)	Use of this method is deprecated. Use simple_merge_p2e instead
	Edge[]	**simple_merge**	(Edge[] in)	Use of this method is deprecated. Use simple_merge_e2e instead
	Edge[]	**simple_merge**	(Edge[] in, int mode)	Use of this method is deprecated. Use simple_merge_e2e instead
	Polygon[]	**simple_merge_to_polygon**	(Polygon[] in, bool resolve_holes, bool min_coherence)	Use of this method is deprecated. Use simple_merge_p2p instead
	Polygon[]	**simple_merge_to_pol**	(Polygon[] in, bool resolve_holes, bool min_coherence, int mode)	Use of this method is deprecated. Use simple_merge_p2p instead
	Polygon[]	**simple_merge_to_polygon**	(Edge[] in, bool resolve_holes, bool min_coherence)	Use of this method is deprecated. Use simple_merge_e2p instead
	Polygon[]	**simple_merge_to_pol**	(Edge[] in, bool resolve_holes, bool min_coherence,	Use of this method is deprecated. Use simple_merge_e2p instead

		int mode)	
Edge[]	size	(Polygon[] in, int dx, int dy, unsigned int mode)	Use of this method is deprecated. Use size_p2e instead
Edge[]	size	(Polygon[] in, int d, unsigned int mode)	Use of this method is deprecated. Use size_p2e instead
Polygon[]	size_to_polygon	(Polygon[] in, int dx, int dy, unsigned int mode, bool resolve_holes, bool min_coherence)	Use of this method is deprecated. Use size_p2p instead
Polygon[]	size_to_polygon	(Polygon[] in, int d, unsigned int mode, bool resolve_holes, bool min_coherence)	Use of this method is deprecated. Use size_p2p instead

Detailed description

ModeANotB

Signature: *[static]* int **ModeANotB**

Description: boolean method's mode value for A NOT B operation

Python specific notes:
The object exposes a readable attribute 'ModeANotB'. This is the getter.

ModeAnd

Signature: *[static]* int **ModeAnd**

Description: boolean method's mode value for AND operation

Python specific notes:
The object exposes a readable attribute 'ModeAnd'. This is the getter.

ModeBNotA

Signature: *[static]* int **ModeBNotA**

Description: boolean method's mode value for B NOT A operation

Python specific notes:
The object exposes a readable attribute 'ModeBNotA'. This is the getter.

ModeOr

Signature: *[static]* int **ModeOr**

Description: boolean method's mode value for OR operation

Python specific notes:
The object exposes a readable attribute 'ModeOr'. This is the getter.

ModeXor

Signature: *[static]* int **ModeXor**

Description: boolean method's mode value for XOR operation

Python specific notes:
The object exposes a readable attribute 'ModeXor'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const EdgeProcessor other)

Description: Assigns another object to self

boolean

(1) Signature: Edge[] **boolean** (Polygon[] a, Polygon[] b, int mode)

Description: Boolean operation for a set of given polygons, creating edges

a:	The input polygons (first operand)
b:	The input polygons (second operand)
mode:	The boolean mode
Returns:	The output edges

Use of this method is deprecated. Use boolean_p2e instead

This method computes the result for the given boolean operation on two sets of polygons. The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

This is a convenience method that bundles filling of the edges, processing with a Boolean operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'boolean'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Edge[] **boolean** (Edge[] a, Edge[] b, int mode)

Description: Boolean operation for a set of given edges, creating edges

a:	The input edges (first operand)
b:	The input edges (second operand)
mode:	The boolean mode (one of the Mode.. values)
Returns:	The output edges

Use of this method is deprecated. Use boolean_e2e instead

This method computes the result for the given boolean operation on two sets of edges. The input edges must form closed contours where holes and hulls must be oriented differently. The input edges are processed with a simple non-zero wrap count rule as a whole.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

Prior to version 0.21 this method was called 'boolean'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

boolean_e2e

Signature: Edge[] **boolean_e2e** (Edge[] a, Edge[] b, int mode)

Description: Boolean operation for a set of given edges, creating edges

a:	The input edges (first operand)
b:	The input edges (second operand)
mode:	The boolean mode (one of the Mode.. values)
Returns:	The output edges

This method computes the result for the given boolean operation on two sets of edges. The input edges must form closed contours where holes and hulls must be oriented differently. The input edges are processed with a simple non-zero wrap count rule as a whole.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

Prior to version 0.21 this method was called 'boolean'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

boolean_e2p

Signature: Polygon[] **boolean_e2p** (Edge[] a, Edge[] b, int mode, bool resolve_holes, bool min_coherence)

Description: Boolean operation for a set of given edges, creating polygons

a:	The input polygons (first operand)
b:	The input polygons (second operand)
mode:	The boolean mode (one of the Mode.. values)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours

Returns: The output polygons

This method computes the result for the given boolean operation on two sets of edges. The input edges must form closed contours where holes and hulls must be oriented differently. The input edges are processed with a simple non-zero wrap count rule as a whole.

This method produces polygons on output and allows fine-tuning of the parameters for that purpose.

Prior to version 0.21 this method was called 'boolean_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

boolean_p2e

Signature: Edge[] **boolean_p2e** (Polygon[] a, Polygon[] b, int mode)

Description: Boolean operation for a set of given polygons, creating edges

a:	The input polygons (first operand)
b:	The input polygons (second operand)
mode:	The boolean mode
Returns:	The output edges

This method computes the result for the given boolean operation on two sets of polygons. The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

This is a convenience method that bundles filling of the edges, processing with a Boolean operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'boolean'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

boolean_p2p

Signature: Polygon[] **boolean_p2p** (Polygon[] a, Polygon[] b, int mode, bool resolve_holes, bool min_coherence)

Description: Boolean operation for a set of given polygons, creating polygons

a:	The input polygons (first operand)
b:	The input polygons (second operand)
mode:	The boolean mode (one of the Mode.. values)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

This method computes the result for the given boolean operation on two sets of polygons. This method produces polygons on output and allows fine-tuning of the parameters for that purpose.

This is a convenience method that bundles filling of the edges, processing with a Boolean operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'boolean_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

boolean_to_polygon

(1) Signature: Polygon[] **boolean_to_polygon** (Polygon[] a, Polygon[] b, int mode, bool resolve_holes, bool min_coherence)

Description: Boolean operation for a set of given polygons, creating polygons

a:	The input polygons (first operand)
b:	The input polygons (second operand)
mode:	The boolean mode (one of the Mode.. values)
resolve_holes:	true, if holes should be resolved into the hull

min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

Use of this method is deprecated. Use boolean_p2p instead

This method computes the result for the given boolean operation on two sets of polygons. This method produces polygons on output and allows fine-tuning of the parameters for that purpose.

This is a convenience method that bundles filling of the edges, processing with a Boolean operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'boolean_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Polygon[] **boolean_to_polygon** (Edge[] a, Edge[] b, int mode, bool resolve_holes, bool min_coherence)

Description: Boolean operation for a set of given edges, creating polygons

a:	The input polygons (first operand)
b:	The input polygons (second operand)
mode:	The boolean mode (one of the Mode.. values)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

Use of this method is deprecated. Use boolean_e2p instead

This method computes the result for the given boolean operation on two sets of edges. The input edges must form closed contours where holes and hulls must be oriented differently. The input edges are processed with a simple non-zero wrap count rule as a whole.

This method produces polygons on output and allows fine-tuning of the parameters for that purpose.

Prior to version 0.21 this method was called 'boolean_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disable_progress

Signature: void **disable_progress**

Description: Disable progress reporting

Calling this method will stop the edge processor from showing a progress bar. See enable_progress.

This method has been introduced in version 0.23.

dup

Signature: *[const]* new EdgeProcessor ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

enable_progress

Signature: void **enable_progress** (string label)

Description: Enable progress reporting

After calling this method, the edge processor will report the progress through a progress bar. The label is a text which is put in front of the progress bar. Using a progress bar will imply a performance penalty of a few percent typically.

This method has been introduced in version 0.23.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

merge

Signature: Edge[] **merge** (Polygon[] in, unsigned int min_wc)

Description: Merge the given polygons

in:	The input polygons
min_wc:	The minimum wrap count for output (0: all polygons, 1: at least two overlapping)
Returns:	The output edges

Use of this method is deprecated. Use merge_p2e instead

In contrast to "simple_merge", this merge implementation considers each polygon individually before merging them. Thus self-overlaps are effectively removed before the output is computed and holes are correctly merged with the hull. In addition, this method allows selecting areas with a higher wrap count which in turn allows computing overlaps of polygons on the same layer. Because this method merges the polygons before the overlap is computed, self-overlapping polygons do not contribute to higher wrap count areas.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

Prior to version 0.21 this method was called 'merge'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

merge_p2e

Signature: Edge[] **merge_p2e** (Polygon[] in, unsigned int min_wc)

Description: Merge the given polygons

in:	The input polygons

min_wc: The minimum wrap count for output (0: all polygons, 1: at least two overlapping)

Returns: The output edges

In contrast to "simple_merge", this merge implementation considers each polygon individually before merging them. Thus self-overlaps are effectively removed before the output is computed and holes are correctly merged with the hull. In addition, this method allows selecting areas with a higher wrap count which in turn allows computing overlaps of polygons on the same layer. Because this method merges the polygons before the overlap is computed, self-overlapping polygons do not contribute to higher wrap count areas.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

Prior to version 0.21 this method was called 'merge'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

merge_p2p

Signature: Polygon[] **merge_p2p** (Polygon[] in, unsigned int min_wc, bool resolve_holes, bool min_coherence)

Description: Merge the given polygons

in: The input polygons

min_wc: The minimum wrap count for output (0: all polygons, 1: at least two overlapping)

resolve_holes: true, if holes should be resolved into the hull

min_coherence: true, if touching corners should be resolved into less connected contours

Returns: The output polygons

In contrast to "simple_merge", this merge implementation considers each polygon individually before merging them. Thus self-overlaps are effectively removed before the output is computed and holes are correctly merged with the hull. In addition, this method allows selecting areas with a higher wrap count which in turn allows computing overlaps of polygons on the same layer. Because this method merges the polygons before the overlap is computed, self-overlapping polygons do not contribute to higher wrap count areas.

This method produces polygons and allows fine-tuning of the parameters for that purpose.

Prior to version 0.21 this method was called 'merge_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

merge_to_polygon

Signature: Polygon[] **merge_to_polygon** (Polygon[] in, unsigned int min_wc, bool resolve_holes, bool min_coherence)

Description: Merge the given polygons

in: The input polygons

min_wc: The minimum wrap count for output (0: all polygons, 1: at least two overlapping)

resolve_holes: true, if holes should be resolved into the hull

min_coherence: true, if touching corners should be resolved into less connected contours

Returns: The output polygons

Use of this method is deprecated. Use merge_p2p instead

In contrast to "simple_merge", this merge implementation considers each polygon individually before merging them. Thus self-overlaps are effectively removed before the output is computed and holes are correctly merged with the hull. In addition, this method allows selecting areas with a higher wrap count which in turn allows computing overlaps of polygons on the same layer. Because this method merges

	the polygons before the overlap is computed, self-overlapping polygons do not contribute to higher wrap count areas. This method produces polygons and allows fine-tuning of the parameters for that purpose. Prior to version 0.21 this method was called 'merge_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.
mode_and	**Signature**: *[static]* int **mode_and** **Description**: boolean method's mode value for AND operation Use of this method is deprecated. Use ModeAnd instead **Python specific notes:** The object exposes a readable attribute 'ModeAnd'. This is the getter.
mode_anotb	**Signature**: *[static]* int **mode_anotb** **Description**: boolean method's mode value for A NOT B operation Use of this method is deprecated. Use ModeANotB instead **Python specific notes:** The object exposes a readable attribute 'ModeANotB'. This is the getter.
mode_bnota	**Signature**: *[static]* int **mode_bnota** **Description**: boolean method's mode value for B NOT A operation Use of this method is deprecated. Use ModeBNotA instead **Python specific notes:** The object exposes a readable attribute 'ModeBNotA'. This is the getter.
mode_or	**Signature**: *[static]* int **mode_or** **Description**: boolean method's mode value for OR operation Use of this method is deprecated. Use ModeOr instead **Python specific notes:** The object exposes a readable attribute 'ModeOr'. This is the getter.
mode_xor	**Signature**: *[static]* int **mode_xor** **Description**: boolean method's mode value for XOR operation Use of this method is deprecated. Use ModeXor instead **Python specific notes:** The object exposes a readable attribute 'ModeXor'. This is the getter.
new	**Signature**: *[static]* new EdgeProcessor ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
simple_merge	**(1) Signature**: Edge[] **simple_merge** (Polygon[] in) **Description**: Merge the given polygons in a simple "non-zero wrapcount" fashion **in**: The input polygons **Returns**: The output edges Use of this method is deprecated. Use simple_merge_p2e instead

The wrapcount is computed over all polygons, i.e. overlapping polygons may "cancel" if they have different orientation (since a polygon is oriented by construction that is not easy to achieve). The other merge operation provided for this purpose is "merge" which normalizes each polygon individually before merging them. "simple_merge" is somewhat faster and consumes less memory.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'simple_merge'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Edge[] **simple_merge** (Polygon[] in, int mode)

Description: Merge the given polygons and specify the merge mode

mode:	See description
in:	The input polygons
Returns:	The output edges

Use of this method is deprecated. Use simple_merge_p2e instead

The wrapcount is computed over all polygons, i.e. overlapping polygons may "cancel" if they have different orientation (since a polygon is oriented by construction that is not easy to achieve). The other merge operation provided for this purpose is "merge" which normalizes each polygon individually before merging them. "simple_merge" is somewhat faster and consumes less memory.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

This method has been added in version 0.22.

The mode specifies the rule to use when producing output. A value of 0 specifies the even-odd rule. A positive value specifies the wrap count threshold (positive only). A negative value specifies the threshold of the absolute value of the wrap count (i.e. -1 is non-zero rule).

(3) Signature: Edge[] **simple_merge** (Edge[] in)

Description: Merge the given edges in a simple "non-zero wrapcount" fashion

in:	The input edges
Returns:	The output edges

Use of this method is deprecated. Use simple_merge_e2e instead

The edges provided must form valid closed contours. Contours oriented differently "cancel" each other. Overlapping contours are merged when the orientation is the same.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'simple_merge'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(4) Signature: Edge[] **simple_merge** (Edge[] in, int mode)

Description: Merge the given polygons and specify the merge mode

mode:	See description
in:	The input edges

Returns: The output edges

Use of this method is deprecated. Use simple_merge_e2e instead

The edges provided must form valid closed contours. Contours oriented differently "cancel" each other. Overlapping contours are merged when the orientation is the same.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

This method has been added in version 0.22.

The mode specifies the rule to use when producing output. A value of 0 specifies the even-odd rule. A positive value specifies the wrap count threshold (positive only). A negative value specifies the threshold of the absolute value of the wrap count (i.e. -1 is non-zero rule).

simple_merge_e2e

(1) Signature: Edge[] **simple_merge_e2e** (Edge[] in)

Description: Merge the given edges in a simple "non-zero wrapcount" fashion

in: The input edges

Returns: The output edges

The edges provided must form valid closed contours. Contours oriented differently "cancel" each other. Overlapping contours are merged when the orientation is the same.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'simple_merge'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Edge[] **simple_merge_e2e** (Edge[] in, int mode)

Description: Merge the given polygons and specify the merge mode

mode: See description

in: The input edges

Returns: The output edges

The edges provided must form valid closed contours. Contours oriented differently "cancel" each other. Overlapping contours are merged when the orientation is the same.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

This method has been added in version 0.22.

The mode specifies the rule to use when producing output. A value of 0 specifies the even-odd rule. A positive value specifies the wrap count threshold (positive only). A negative value specifies the threshold of the absolute value of the wrap count (i.e. -1 is non-zero rule).

simple_merge_e2p

(1) Signature: Polygon[] **simple_merge_e2p** (Edge[] in, bool resolve_holes, bool min_coherence)

Description: Merge the given edges in a simple "non-zero wrapcount" fashion into polygons

in: The input edges

resolve_holes: true, if holes should be resolved into the hull

min_coherence: true, if touching corners should be resolved into less connected contours

Returns: The output polygons

The edges provided must form valid closed contours. Contours oriented differently "cancel" each other. Overlapping contours are merged when the orientation is the same.

This method produces polygons and allows fine-tuning of the parameters for that purpose.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'simple_merge_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Polygon[] **simple_merge_e2p** (Edge[] in, bool resolve_holes, bool min_coherence, int mode)

Description: Merge the given polygons and specify the merge mode

mode: See description

in: The input edges

resolve_holes: true, if holes should be resolved into the hull

min_coherence: true, if touching corners should be resolved into less connected contours

Returns: The output polygons

The edges provided must form valid closed contours. Contours oriented differently "cancel" each other. Overlapping contours are merged when the orientation is the same.

This method produces polygons and allows fine-tuning of the parameters for that purpose.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

This method has been added in version 0.22.

The mode specifies the rule to use when producing output. A value of 0 specifies the even-odd rule. A positive value specifies the wrap count threshold (positive only). A negative value specifies the threshold of the absolute value of the wrap count (i.e. -1 is non-zero rule).

simple_merge_p2e

(1) Signature: Edge[] **simple_merge_p2e** (Polygon[] in)

Description: Merge the given polygons in a simple "non-zero wrapcount" fashion

in: The input polygons

Returns: The output edges

The wrapcount is computed over all polygons, i.e. overlapping polygons may "cancel" if they have different orientation (since a polygon is oriented by construction that is not easy to achieve). The other merge operation provided for this purpose is "merge" which normalizes each polygon individually before merging them. "simple_merge" is somewhat faster and consumes less memory.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'simple_merge'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Edge[] **simple_merge_p2e** (Polygon[] in, int mode)

Description: Merge the given polygons and specify the merge mode

mode:	See description
in:	The input polygons
Returns:	The output edges

The wrapcount is computed over all polygons, i.e. overlapping polygons may "cancel" if they have different orientation (since a polygon is oriented by construction that is not easy to achieve). The other merge operation provided for this purpose is "merge" which normalizes each polygon individually before merging them. "simple_merge" is somewhat faster and consumes less memory.

The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

This method has been added in version 0.22.

The mode specifies the rule to use when producing output. A value of 0 specifies the even-odd rule. A positive value specifies the wrap count threshold (positive only). A negative value specifies the threshold of the absolute value of the wrap count (i.e. -1 is non-zero rule).

simple_merge_p2p

(1) Signature: Polygon[] **simple_merge_p2p** (Polygon[] in, bool resolve_holes, bool min_coherence)

Description: Merge the given polygons in a simple "non-zero wrapcount" fashion into polygons

in:	The input polygons
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

The wrapcount is computed over all polygons, i.e. overlapping polygons may "cancel" if they have different orientation (since a polygon is oriented by construction that is not easy to achieve). The other merge operation provided for this purpose is "merge" which normalizes each polygon individually before merging them. "simple_merge" is somewhat faster and consumes less memory.

This method produces polygons and allows fine-tuning of the parameters for that purpose.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'simple_merge_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Polygon[] **simple_merge_p2p** (Polygon[] in, bool resolve_holes, bool min_coherence, int mode)

Description: Merge the given polygons and specify the merge mode

mode:	See description
in:	The input polygons
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

The wrapcount is computed over all polygons, i.e. overlapping polygons may "cancel" if they have different orientation (since a polygon is oriented by construction that is not easy to achieve). The other merge operation provided for this purpose is "merge" which normalizes each polygon individually before merging them. "simple_merge" is somewhat faster and consumes less memory.

This method produces polygons and allows fine-tuning of the parameters for that purpose.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

This method has been added in version 0.22.

The mode specifies the rule to use when producing output. A value of 0 specifies the even-odd rule. A positive value specifies the wrap count threshold (positive only). A negative value specifies the threshold of the absolute value of the wrap count (i.e. -1 is non-zero rule).

simple_merge_to_polygon

(1) Signature: Polygon[] **simple_merge_to_polygon** (Polygon[] in, bool resolve_holes, bool min_coherence)

Description: Merge the given polygons in a simple "non-zero wrapcount" fashion into polygons

in:	The input polygons
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

Use of this method is deprecated. Use simple_merge_p2p instead

The wrapcount is computed over all polygons, i.e. overlapping polygons may "cancel" if they have different orientation (since a polygon is oriented by construction that is not easy to achieve). The other merge operation provided for this purpose is "merge" which normalizes each polygon individually before merging them. "simple_merge" is somewhat faster and consumes less memory.

This method produces polygons and allows fine-tuning of the parameters for that purpose.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'simple_merge_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Polygon[] **simple_merge_to_polygon** (Polygon[] in, bool resolve_holes, bool min_coherence, int mode)

Description: Merge the given polygons and specify the merge mode

mode:	See description
in:	The input polygons
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

Use of this method is deprecated. Use simple_merge_p2p instead

The wrapcount is computed over all polygons, i.e. overlapping polygons may "cancel" if they have different orientation (since a polygon is oriented by construction that is not easy to achieve). The other merge operation provided for this purpose is "merge" which normalizes each polygon individually before merging them. "simple_merge" is somewhat faster and consumes less memory.

This method produces polygons and allows fine-tuning of the parameters for that purpose.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

This method has been added in version 0.22.

The mode specifies the rule to use when producing output. A value of 0 specifies the even-odd rule. A positive value specifies the wrap count threshold (positive only). A negative value specifies the threshold of the absolute value of the wrap count (i.e. -1 is non-zero rule).

(3) Signature: Polygon[] **simple_merge_to_polygon** (Edge[] in, bool resolve_holes, bool min_coherence)

Description: Merge the given edges in a simple "non-zero wrapcount" fashion into polygons

in:	The input edges
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

Use of this method is deprecated. Use simple_merge_e2p instead

The edges provided must form valid closed contours. Contours oriented differently "cancel" each other. Overlapping contours are merged when the orientation is the same.

This method produces polygons and allows fine-tuning of the parameters for that purpose.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

Prior to version 0.21 this method was called 'simple_merge_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(4) Signature: Polygon[] **simple_merge_to_polygon** (Edge[] in, bool resolve_holes, bool min_coherence, int mode)

Description: Merge the given polygons and specify the merge mode

mode:	See description
in:	The input edges
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

Use of this method is deprecated. Use simple_merge_e2p instead

The edges provided must form valid closed contours. Contours oriented differently "cancel" each other. Overlapping contours are merged when the orientation is the same.

This method produces polygons and allows fine-tuning of the parameters for that purpose.

This is a convenience method that bundles filling of the edges, processing with a SimpleMerge operator and puts the result into an output vector.

This method has been added in version 0.22.

The mode specifies the rule to use when producing output. A value of 0 specifies the even-odd rule. A positive value specifies the wrap count threshold (positive only). A negative value specifies the threshold of the absolute value of the wrap count (i.e. -1 is non-zero rule).

size

(1) Signature: Edge[] **size** (Polygon[] in, int dx, int dy, unsigned int mode)

Description: Size the given polygons

in:	The input polygons
dx:	The sizing value in x direction
dy:	The sizing value in y direction
mode:	The sizing mode (standard is 2)
Returns:	The output edges

Use of this method is deprecated. Use size_p2e instead

This method sizes a set of polygons. Before the sizing is applied, the polygons are merged. After that, sizing is applied on the individual result polygons of the merge step. The result may contain overlapping contours, but no self-overlaps.

dx and dy describe the sizing. A positive value indicates oversize (outwards) while a negative one describes undersize (inwards). The sizing applied can be chosen differently in x and y direction. In this case, the sign must be identical for both dx and dy.

The 'mode' parameter describes the corner fill strategy. Mode 0 connects all corner segments directly. Mode 1 is the 'octagon' strategy in which square corners are interpolated with a partial octagon. Mode 2 is the standard mode in which corners are filled by expanding edges unless these edges form a sharp bend with an angle of more than 90 degree. In that case, the corners are cut off. In Mode 3, no cutoff occurs up to a bending angle of 135 degree. Mode 4 and 5 are even more aggressive and allow very sharp bends without cutoff. This strategy may produce long spikes on sharply bending corners. The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

Prior to version 0.21 this method was called 'size'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Edge[] **size** (Polygon[] in, int d, unsigned int mode)

Description: Size the given polygons (isotropic)

in:	The input polygons
d:	The sizing value in x direction
mode:	The sizing mode
Returns:	The output edges

Use of this method is deprecated. Use size_p2e instead

This method is equivalent to calling the anisotropic version with identical dx and dy.

Prior to version 0.21 this method was called 'size'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

size_p2e

(1) Signature: Edge[] **size_p2e** (Polygon[] in, int dx, int dy, unsigned int mode)

Description: Size the given polygons

in:	The input polygons
dx:	The sizing value in x direction
dy:	The sizing value in y direction
mode:	The sizing mode (standard is 2)
Returns:	The output edges

This method sizes a set of polygons. Before the sizing is applied, the polygons are merged. After that, sizing is applied on the individual result polygons of the merge step. The result may contain overlapping contours, but no self-overlaps.

dx and dy describe the sizing. A positive value indicates oversize (outwards) while a negative one describes undersize (inwards). The sizing applied can be chosen differently in x and y direction. In this case, the sign must be identical for both dx and dy.

The 'mode' parameter describes the corner fill strategy. Mode 0 connects all corner segments directly. Mode 1 is the 'octagon' strategy in which square corners are interpolated with a partial octagon. Mode 2 is the standard mode in which corners are filled by expanding edges unless these edges form a sharp bend with an angle of more than 90 degree. In that case, the corners are cut off. In Mode 3, no cutoff occurs up to a bending angle of 135 degree. Mode 4 and 5 are even more aggressive and allow very sharp bends without cutoff. This strategy may produce long spikes on sharply bending corners. The result is presented as a set of edges forming closed contours. Hulls are oriented clockwise while holes are oriented counter-clockwise.

Prior to version 0.21 this method was called 'size'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Edge[] **size_p2e** (Polygon[] in, int d, unsigned int mode)

Description: Size the given polygons (isotropic)

in:	The input polygons
d:	The sizing value in x direction
mode:	The sizing mode
Returns:	The output edges

This method is equivalent to calling the anisotropic version with identical dx and dy.

Prior to version 0.21 this method was called 'size'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

size_p2p

(1) Signature: Polygon[] **size_p2p** (Polygon[] in, int dx, int dy, unsigned int mode, bool resolve_holes, bool min_coherence)

Description: Size the given polygons into polygons

in:	The input polygons
dx:	The sizing value in x direction
dy:	The sizing value in y direction
mode:	The sizing mode (standard is 2)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

This method sizes a set of polygons. Before the sizing is applied, the polygons are merged. After that, sizing is applied on the individual result polygons of the merge step. The result may contain overlapping polygons, but no self-overlapping ones. Polygon overlap occurs if the polygons are close enough, so a positive sizing makes polygons overlap.

dx and dy describe the sizing. A positive value indicates oversize (outwards) while a negative one describes undersize (inwards). The sizing applied can be chosen differently in x and y direction. In this case, the sign must be identical for both dx and dy.

The 'mode' parameter describes the corner fill strategy. Mode 0 connects all corner segments directly. Mode 1 is the 'octagon' strategy in which square corners are interpolated with a partial octagon. Mode 2 is the standard mode in which corners are filled by expanding edges unless these edges form a sharp bend with an angle of more than 90 degree. In that case, the corners are cut off. In Mode 3, no cutoff occurs up to a bending angle of 135 degree. Mode 4 and 5 are even more aggressive and allow very sharp bends without cutoff. This strategy may produce long spikes on sharply bending corners. This method produces polygons and allows fine-tuning of the parameters for that purpose.

Prior to version 0.21 this method was called 'size_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Polygon[] **size_p2p** (Polygon[] in, int d, unsigned int mode, bool resolve_holes, bool min_coherence)

Description: Size the given polygons into polygons (isotropic)

in:	The input polygons
d:	The sizing value in x direction
mode:	The sizing mode
resolve_holes:	true, if holes should be resolved into the hull

min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

This method is equivalent to calling the anisotropic version with identical dx and dy.

Prior to version 0.21 this method was called 'size_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

size_to_polygon

(1) Signature: Polygon[] **size_to_polygon** (Polygon[] in, int dx, int dy, unsigned int mode, bool resolve_holes, bool min_coherence)

Description: Size the given polygons into polygons

in:	The input polygons
dx:	The sizing value in x direction
dy:	The sizing value in y direction
mode:	The sizing mode (standard is 2)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

Use of this method is deprecated. Use size_p2p instead

This method sizes a set of polygons. Before the sizing is applied, the polygons are merged. After that, sizing is applied on the individual result polygons of the merge step. The result may contain overlapping polygons, but no self-overlapping ones. Polygon overlap occurs if the polygons are close enough, so a positive sizing makes polygons overlap.

dx and dy describe the sizing. A positive value indicates oversize (outwards) while a negative one describes undersize (inwards). The sizing applied can be chosen differently in x and y direction. In this case, the sign must be identical for both dx and dy.

The 'mode' parameter describes the corner fill strategy. Mode 0 connects all corner segments directly. Mode 1 is the 'octagon' strategy in which square corners are interpolated with a partial octagon. Mode 2 is the standard mode in which corners are filled by expanding edges unless these edges form a sharp bend with an angle of more than 90 degree. In that case, the corners are cut off. In Mode 3, no cutoff occurs up to a bending angle of 135 degree. Mode 4 and 5 are even more aggressive and allow very sharp bends without cutoff. This strategy may produce long spikes on sharply bending corners. This method produces polygons and allows fine-tuning of the parameters for that purpose.

Prior to version 0.21 this method was called 'size_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

(2) Signature: Polygon[] **size_to_polygon** (Polygon[] in, int d, unsigned int mode, bool resolve_holes, bool min_coherence)

Description: Size the given polygons into polygons (isotropic)

in:	The input polygons
d:	The sizing value in x direction
mode:	The sizing mode
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if touching corners should be resolved into less connected contours
Returns:	The output polygons

Use of this method is deprecated. Use size_p2p instead

This method is equivalent to calling the anisotropic version with identical dx and dy.

Prior to version 0.21 this method was called 'size_to_polygon'. Is was renamed to avoid ambiguities for empty input arrays. The old version is still available but deprecated.

4.43. API reference - Class EdgeFilter

Notation used in Ruby API documentation

Module: db

Description: A generic edge filter adaptor

Edge filters are an efficient way to filter edge from a Edges collection. To apply a filter, derive your own filter class and pass an instance to the Edges#filter or Edges#filtered method.

Conceptually, these methods take each edge from the collection and present it to the filter's 'selected' method. Based on the result of this evaluation, the edge is kept or discarded.

The magic happens when deep mode edge collections are involved. In that case, the filter will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the filter behaves. You need to configure the filter by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using the filter.

You can skip this step, but the filter algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

Here is some example that filters edges parallel to a given one:

```
class ParallelFilter < RBA::EdgeFilter

  # Constructor
  def initialize(ref_edge)
    self.is_scale_invariant   # orientation matters, but scale does not
    @ref_edge = ref_edge
  end

  # Select only parallel ones
  def selected(edge)
    return edge.is_parallel?(@ref_edge)
  end

end

edges = ... # some Edges object
ref_edge = ... # some Edge
parallel_only = edges.filtered(ParallelFilter::new(ref_edge))
```

This class has been introduced in version 0.29.

Public constructors

new EdgeFilter ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.

	void	_unmanage		Marks the object as no longer owned by the script side.
	void	is_isotropic		Indicates that the filter has isotropic properties
	void	is_isotropic_and_scale_invariant		Indicates that the filter is isotropic and scale invariant
	void	is_scale_invariant		Indicates that the filter is scale invariant
	void	requires_raw_input=	(bool flag)	Sets a value indicating whether the filter needs raw (unmerged) input
[const]	bool	requires_raw_input?		Gets a value indicating whether the filter needs raw (unmerged) input
[virtual,const]	bool	selected	(const Edge edge)	Selects an edge
	void	wants_variants=	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	wants_variants?		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) filters are area or perimeter filters. The area or perimeter of a polygon depends on the scale, but not on the orientation of the polygon.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) filter is the square selector. Whether a polygon is a square or not does not depend on the polygon's orientation nor scale.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) filter is the bounding box aspect ratio (height/width) filter. The definition of heigh and width depends on the orientation, but the ratio is independent on scale.

new

Signature: *[static]* new EdgeFilter ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

requires_raw_input=

Signature: void **requires_raw_input=** (bool flag)

Description: Sets a value indicating whether the filter needs raw (unmerged) input

This flag must be set before using this filter. It tells the filter implementation whether the filter wants to have raw input (unmerged). The default value is 'false', meaning that the filter will receive merged polygons ('merged semantics').

Setting this value to false potentially saves some CPU time needed for merging the polygons. Also, raw input means that strange shapes such as dot-like edges, self-overlapping polygons, empty or degenerated polygons are preserved.

Python specific notes:
The object exposes a writable attribute 'requires_raw_input'. This is the setter.

requires_raw_input?

Signature: *[const]* bool **requires_raw_input?**

Description: Gets a value indicating whether the filter needs raw (unmerged) input

See requires_raw_input= for details.

Python specific notes:
The object exposes a readable attribute 'requires_raw_input'. This is the getter.

selected

Signature: *[virtual,const]* bool **selected** (const Edge edge)

Description: Selects an edge

This method is the actual payload. It needs to be reimplemented in a derived class. It needs to analyze the edge and return 'true' if it should be kept and 'false' if it should be discarded.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.44. API reference - Class EdgeOperator

Notation used in Ruby API documentation

Module: db

Description: A generic edge-to-polygon operator

Edge processors are an efficient way to process edges from an edge collection. To apply a processor, derive your own operator class and pass an instance to the Edges#processed method.

Conceptually, these methods take each edge from the edge collection and present it to the operator's 'process' method. The result of this call is a list of zero to many output edges derived from the input edge. The output edge collection is the sum over all these individual results.

The magic happens when deep mode edge collections are involved. In that case, the processor will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the operator behaves. You need to configure the operator by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using it.

You can skip this step, but the processor algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

Here is some example that shrinks every edge to half of the size, but does not change the position. In this example the 'position' is defined by the center of the edge:

```
class ShrinkToHalf < RBA::EdgeOperator

  # Constructor
  def initialize
    self.is_isotropic_and_scale_invariant   # scale or orientation do not matter
  end

  # Shrink to half size
  def process(edge)
    shift = edge.bbox.center - RBA::Point::new   # shift vector
    return [ (edge.moved(-shift) * 0.5).moved(shift) ]
  end

end

edges = ... # some Edges collection
shrinked_to_half = edges.processed(ShrinkToHalf::new)
```

This class has been introduced in version 0.29.

Public constructors

new EdgeOperator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference

	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
	void	is_isotropic		Indicates that the filter has isotropic properties
	void	is_isotropic_and_scale_invariant		Indicates that the filter is isotropic and scale invariant
	void	is_scale_invariant		Indicates that the filter is scale invariant
[virtual,const]	Edge[]	process	(const Edge shape)	Processes a shape
	void	requires_raw_input=	(bool flag)	Sets a value indicating whether the processor needs raw (unmerged) input
[const]	bool	requires_raw_input?		Gets a value indicating whether the processor needs raw (unmerged) input
	void	result_is_merged=	(bool flag)	Sets a value indicating whether the processor delivers merged output
[const]	bool	result_is_merged?		Gets a value indicating whether the processor delivers merged output
	void	result_must_not_be_merge	(bool flag)	Sets a value indicating whether the processor's output must not be merged
[const]	bool	result_must_not_be_merged?		Gets a value indicating whether the processor's output must not be merged
	void	wants_variants=	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	wants_variants?		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) processors are size or shrink operators. Size or shrink is not dependent on orientation unless size or shrink needs to be different in x and y direction.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) processor is the convex decomposition operator. The decomposition of a polygon into convex parts is an operation that is not depending on scale nor orientation.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) processor is the rotation operator. Rotation is not depending on scale, but on the original orientation as mirrored versions need to be rotated differently.

new

Signature: *[static]* new EdgeOperator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

process

Signature: *[virtual,const]* Edge[] **process** (const Edge shape)

Description: Processes a shape

	This method is the actual payload. It needs to be reimplemented in a derived class. If needs to process the input shape and deliver a list of output shapes. The output list may be empty to entirely discard the input shape. It may also contain more than a single shape. In that case, the number of total shapes may grow during application of the processor.
requires_raw_input=	**Signature**: void **requires_raw_input=** (bool flag) **Description**: Sets a value indicating whether the processor needs raw (unmerged) input This flag must be set before using this processor. It tells the processor implementation whether the processor wants to have raw input (unmerged). The default value is 'false', meaning that the processor will receive merged polygons ('merged semantics'). Setting this value to false potentially saves some CPU time needed for merging the polygons. Also, raw input means that strange shapes such as dot-like edges, self-overlapping polygons, empty or degenerated polygons are preserved. **Python specific notes:** The object exposes a writable attribute 'requires_raw_input'. This is the setter.
requires_raw_input?	**Signature**: *[const]* bool **requires_raw_input?** **Description**: Gets a value indicating whether the processor needs raw (unmerged) input See requires_raw_input= for details. **Python specific notes:** The object exposes a readable attribute 'requires_raw_input'. This is the getter.
result_is_merged=	**Signature**: void **result_is_merged=** (bool flag) **Description**: Sets a value indicating whether the processor delivers merged output This flag must be set before using this processor. If the processor maintains the merged condition by design (output is merged if input is), it is a good idea to set this predicate to 'true'. This will avoid additional merge steps when the resulting collection is used in further operations that need merged input . **Python specific notes:** The object exposes a writable attribute 'result_is_merged'. This is the setter.
result_is_merged?	**Signature**: *[const]* bool **result_is_merged?** **Description**: Gets a value indicating whether the processor delivers merged output See result_is_merged= for details. **Python specific notes:** The object exposes a readable attribute 'result_is_merged'. This is the getter.
result_must_not_be_merged=	**Signature**: void **result_must_not_be_merged=** (bool flag) **Description**: Sets a value indicating whether the processor's output must not be merged This flag must be set before using this processor. The processor can set this flag if it wants to deliver shapes that must not be merged - e.g. point-like edges or strange or degenerated polygons. . **Python specific notes:** The object exposes a writable attribute 'result_must_not_be_merged'. This is the setter.
result_must_not_be_merged	**Signature**: *[const]* bool **result_must_not_be_merged?** **Description**: Gets a value indicating whether the processor's output must not be merged See result_must_not_be_merged= for details. **Python specific notes:**

	The object exposes a readable attribute 'result_must_not_be_merged'. This is the getter.
wants_variants=	**Signature**: void **wants_variants=** (bool flag) **Description**: Sets a value indicating whether the filter prefers cell variants This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false). This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified. **Python specific notes:** The object exposes a writable attribute 'wants_variants'. This is the setter.
wants_variants?	**Signature**: *[const]* bool **wants_variants?** **Description**: Gets a value indicating whether the filter prefers cell variants See wants_variants= for details. **Python specific notes:** The object exposes a readable attribute 'wants_variants'. This is the getter.

4.45. API reference - Class EdgeToPolygonOperator

Notation used in Ruby API documentation

Module: db

Description: A generic edge-to-polygon operator

Edge processors are an efficient way to process edges from an edge collection. To apply a processor, derive your own operator class and pass an instance to the Edges#processed method.

Conceptually, these methods take each edge from the edge collection and present it to the operator's 'process' method. The result of this call is a list of zero to many output polygons derived from the input edge. The output region is the sum over all these individual results.

The magic happens when deep mode edge collections are involved. In that case, the processor will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the operator behaves. You need to configure the operator by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using it.

You can skip this step, but the processor algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

For a basic example see the EdgeOperator class, with the exception that this incarnation has to deliver edges.

This class has been introduced in version 0.29.

Public constructors

new EdgeToPolygonOperator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**is_isotropic**		Indicates that the filter has isotropic properties
	void	**is_isotropic_and_scale_invariant**		Indicates that the filter is isotropic and scale invariant
	void	**is_scale_invariant**		Indicates that the filter is scale invariant
[virtual,const]	Polygon[]	**process**	(const Edge shape)	Processes a shape
	void	**requires_raw_input=**	(bool flag)	Sets a value indicating whether the processor needs raw (unmerged) input

[const]	bool	**requires_raw_input?**		Gets a value indicating whether the processor needs raw (unmerged) input
	void	**result_is_merged=**	(bool flag)	Sets a value indicating whether the processor delivers merged output
[const]	bool	**result_is_merged?**		Gets a value indicating whether the processor delivers merged output
	void	**result_must_not_be_merge**	(bool flag)	Sets a value indicating whether the processor's output must not be merged
[const]	bool	**result_must_not_be_merged?**		Gets a value indicating whether the processor's output must not be merged
	void	**wants_variants=**	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	**wants_variants?**		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) processors are size or shrink operators. Size or shrink is not dependent on orientation unless size or shrink needs to be different in x and y direction.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) processor is the convex decomposition operator. The decomposition of a polygon into convex parts is an operation that is not depending on scale nor orientation.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) processor is the rotation operator. Rotation is not depending on scale, but on the original orientation as mirrored versions need to be rotated differently.

new

Signature: *[static]* new EdgeToPolygonOperator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

process

Signature: *[virtual,const]* Polygon[] **process** (const Edge shape)

Description: Processes a shape

This method is the actual payload. It needs to be reimplemented in a derived class. If needs to process the input shape and deliver a list of output shapes. The output list may be empty to entirely discard the input shape. It may also contain more than a single shape. In that case, the number of total shapes may grow during application of the processor.

requires_raw_input=

Signature: void **requires_raw_input=** (bool flag)

Description: Sets a value indicating whether the processor needs raw (unmerged) input

This flag must be set before using this processor. It tells the processor implementation whether the processor wants to have raw input (unmerged). The default value is 'false', meaning that the processor will receive merged polygons ('merged semantics').

Setting this value to false potentially saves some CPU time needed for merging the polygons. Also, raw input means that strange shapes such as dot-like edges, self-overlapping polygons, empty or degenerated polygons are preserved.

Python specific notes:
The object exposes a writable attribute 'requires_raw_input'. This is the setter.

requires_raw_input?

Signature: *[const]* bool **requires_raw_input?**

Description: Gets a value indicating whether the processor needs raw (unmerged) input

See requires_raw_input= for details.

Python specific notes:
The object exposes a readable attribute 'requires_raw_input'. This is the getter.

result_is_merged=

Signature: void **result_is_merged=** (bool flag)

Description: Sets a value indicating whether the processor delivers merged output

This flag must be set before using this processor. If the processor maintains the merged condition by design (output is merged if input is), it is a good idea to set this predicate to 'true'. This will avoid additional merge steps when the resulting collection is used in further operations that need merged input .

Python specific notes:
The object exposes a writable attribute 'result_is_merged'. This is the setter.

result_is_merged?

Signature: *[const]* bool **result_is_merged?**

Description: Gets a value indicating whether the processor delivers merged output

See result_is_merged= for details.

Python specific notes:
The object exposes a readable attribute 'result_is_merged'. This is the getter.

result_must_not_be_merged=

Signature: void **result_must_not_be_merged=** (bool flag)

Description: Sets a value indicating whether the processor's output must not be merged

This flag must be set before using this processor. The processor can set this flag if it wants to deliver shapes that must not be merged - e.g. point-like edges or strange or degenerated polygons. .

Python specific notes:
The object exposes a writable attribute 'result_must_not_be_merged'. This is the setter.

result_must_not_be_merged

Signature: *[const]* bool **result_must_not_be_merged?**

Description: Gets a value indicating whether the processor's output must not be merged

See result_must_not_be_merged= for details.

Python specific notes:
The object exposes a readable attribute 'result_must_not_be_merged'. This is the getter.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.46. API reference - Class EdgeToEdgePairOperator

Notation used in Ruby API documentation

Module: db

Description: A generic edge-to-edge-pair operator

Edge processors are an efficient way to process edges from an edge collection. To apply a processor, derive your own operator class and pass an instance to the Edges#processed method.

Conceptually, these methods take each edge from the edge collection and present it to the operator's 'process' method. The result of this call is a list of zero to many output edge pairs derived from the input edge. The output edge pair collection is the sum over all these individual results.

The magic happens when deep mode edge collections are involved. In that case, the processor will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the operator behaves. You need to configure the operator by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using it.

You can skip this step, but the processor algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

For a basic example see the EdgeOperator class, with the exception that this incarnation has to deliver edge pairs.

This class has been introduced in version 0.29.

Public constructors

new EdgeToEdgePairOperator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**is_isotropic**		Indicates that the filter has isotropic properties
	void	**is_isotropic_and_scale_invariant**		Indicates that the filter is isotropic and scale invariant
	void	**is_scale_invariant**		Indicates that the filter is scale invariant
[virtual,const]	EdgePair[]	**process**	(const Edge shape)	Processes a shape
	void	**requires_raw_input=**	(bool flag)	Sets a value indicating whether the processor needs raw (unmerged) input

[const]	bool	**requires_raw_input?**		Gets a value indicating whether the processor needs raw (unmerged) input
	void	**result_is_merged=**	(bool flag)	Sets a value indicating whether the processor delivers merged output
[const]	bool	**result_is_merged?**		Gets a value indicating whether the processor delivers merged output
	void	**result_must_not_be_merge**	(bool flag)	Sets a value indicating whether the processor's output must not be merged
[const]	bool	**result_must_not_be_merged?**		Gets a value indicating whether the processor's output must not be merged
	void	**wants_variants=**	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	**wants_variants?**		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) processors are size or shrink operators. Size or shrink is not dependent on orientation unless size or shrink needs to be different in x and y direction.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) processor is the convex decomposition operator. The decomposition of a polygon into convex parts is an operation that is not depending on scale nor orientation.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) processor is the rotation operator. Rotation is not depending on scale, but on the original orientation as mirrored versions need to be rotated differently.

new

Signature: *[static]* new EdgeToEdgePairOperator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

process

Signature: *[virtual,const]* EdgePair[] **process** (const Edge shape)

Description: Processes a shape

This method is the actual payload. It needs to be reimplemented in a derived class. If needs to process the input shape and deliver a list of output shapes. The output list may be empty to entirely discard the input shape. It may also contain more than a single shape. In that case, the number of total shapes may grow during application of the processor.

requires_raw_input=

Signature: void **requires_raw_input=** (bool flag)

Description: Sets a value indicating whether the processor needs raw (unmerged) input

This flag must be set before using this processor. It tells the processor implementation whether the processor wants to have raw input (unmerged). The default value is 'false', meaning that the processor will receive merged polygons ('merged semantics').

Setting this value to false potentially saves some CPU time needed for merging the polygons. Also, raw input means that strange shapes such as dot-like edges, self-overlapping polygons, empty or degenerated polygons are preserved.

Python specific notes:
The object exposes a writable attribute 'requires_raw_input'. This is the setter.

requires_raw_input?

Signature: *[const]* bool **requires_raw_input?**

Description: Gets a value indicating whether the processor needs raw (unmerged) input

See requires_raw_input= for details.

Python specific notes:
The object exposes a readable attribute 'requires_raw_input'. This is the getter.

result_is_merged=

Signature: void **result_is_merged=** (bool flag)

Description: Sets a value indicating whether the processor delivers merged output

This flag must be set before using this processor. If the processor maintains the merged condition by design (output is merged if input is), it is a good idea to set this predicate to 'true'. This will avoid additional merge steps when the resulting collection is used in further operations that need merged input .

Python specific notes:
The object exposes a writable attribute 'result_is_merged'. This is the setter.

result_is_merged?

Signature: *[const]* bool **result_is_merged?**

Description: Gets a value indicating whether the processor delivers merged output

See result_is_merged= for details.

Python specific notes:
The object exposes a readable attribute 'result_is_merged'. This is the getter.

result_must_not_be_merged=

Signature: void **result_must_not_be_merged=** (bool flag)

Description: Sets a value indicating whether the processor's output must not be merged

This flag must be set before using this processor. The processor can set this flag if it wants to deliver shapes that must not be merged - e.g. point-like edges or strange or degenerated polygons. .

Python specific notes:
The object exposes a writable attribute 'result_must_not_be_merged'. This is the setter.

result_must_not_be_merged

Signature: *[const]* bool **result_must_not_be_merged?**

Description: Gets a value indicating whether the processor's output must not be merged

See result_must_not_be_merged= for details.

Python specific notes:
The object exposes a readable attribute 'result_must_not_be_merged'. This is the getter.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.47. API reference - Class Edges

Notation used in Ruby API documentation

Module: db

Description: A collection of edges (Not necessarily describing closed contours)

Class hierarchy: Edges » ShapeCollection

Sub-classes: EdgeType

This class was introduced to simplify operations on edges sets. See Edge for a description of the individual edge object. The edge collection contains an arbitrary number of edges and supports operations to select edges by various criteria, produce polygons from the edges by applying an extension, filtering edges against other edges collections and checking geometrical relations to other edges (DRC functionality).

The edge collection is supposed to work closely with the Region polygon set. Both are related, although the edge collection has a lower rank since it potentially represents a disconnected collection of edges. Edge collections may form closed contours, for example immediately after they have been derived from a polygon set using Region#edges. But this state is volatile and can easily be destroyed by filtering edges. Hence the connected state does not play an important role in the edge collection's API.

Edge collections may also contain points (degenerated edges with identical start and end points). Such point-like objects participate in some although not all methods of the edge collection class. Edge collections can be used in two different flavors: in raw mode or merged semantics. With merged semantics (the default), connected edges are considered to belong together and are effectively merged. Overlapping parts are counted once in that mode. Dot-like edges are not considered in merged semantics. In raw mode (without merged semantics), each edge is considered as it is. Overlaps between edges may exists and merging has to be done explicitly using the merge method. The semantics can be selected using merged_semantics=.

This class has been introduced in version 0.23.

Public constructors

new Edges ptr	**new**		Default constructor
new Edges ptr	**new**	(const Edge edge)	Constructor from a single edge
new Edges ptr	**new**	(Polygon[] array)	Constructor from a polygon array
new Edges ptr	**new**	(Edge[] array)	Constructor from an edge array
new Edges ptr	**new**	(const Box box)	Box constructor
new Edges ptr	**new**	(const Polygon polygon)	Polygon constructor
new Edges ptr	**new**	(const SimplePolygon polygon)	Simple polygon constructor
new Edges ptr	**new**	(const Path path)	Path constructor
new Edges ptr	**new**	(const Shapes shapes, bool as_edges = true)	Constructor of a flat edge collection from a Shapes container
new Edges ptr	**new**	(const RecursiveShapeIterator shape_iterator, bool as_edges = true)	Constructor of a flat edge collection from a hierarchical shape set
new Edges ptr	**new**	(const RecursiveShapeIterator shape_iterator, const ICplxTrans trans, bool as_edges = true)	Constructor of a flat edge collection from a hierarchical shape set with a transformation

new Edges ptr	new	(const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, bool as_edges = true)	Constructor of a hierarchical edge collection
new Edges ptr	new	(const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, const ICplxTrans trans, bool as_edges = true)	Constructor of a hierarchical edge collection with a transformation
new Edges ptr	new	(const RecursiveShapeIterator shape_iterator, string expr, bool as_pattern = true)	Constructor from a text set
new Edges ptr	new	(const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, string expr, bool as_pattern = true)	Constructor from a text set

Public methods

[const]	Edges	&	(const Edges other)	Returns the boolean AND between self and the other edge collection
[const]	Edges	&	(const Region other)	Returns the parts of the edges inside the given region
	Edges	&=	(const Edges other)	Performs the boolean AND between self and the other edge collection in-place (modifying self)
	Edges	&=	(const Region other)	Selects the parts of the edges inside the given region in-place (modifying self)
[const]	Edges	+	(const Edges other)	Returns the combined edge set of self and the other one
	Edges	+=	(const Edges other)	Adds the edges of the other edge collection to self
[const]	Edges	-	(const Edges other)	Returns the boolean NOT between self and the other edge collection
[const]	Edges	-	(const Region other)	Returns the parts of the edges outside the given region
	Edges	-=	(const Edges other)	Performs the boolean NOT between self and the other edge collection in-place (modifying self)
	Edges	-=	(const Region other)	Selects the parts of the edges outside the given region in-place (modifying self)
[const]	const Edge ptr	[]	(unsigned long n)	Returns the nth edge of the collection

[const]	Edges	^	(const Edges other)	Returns the boolean XOR between self and the other edge collection
	Edges	^=	(const Edges other)	Performs the boolean XOR between self and the other edge collection in-place (modifying self)
	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	Edges	and	(const Edges other)	Returns the boolean AND between self and the other edge collection
[const]	Edges	and	(const Region other)	Returns the parts of the edges inside the given region
	Edges	and_with	(const Edges other)	Performs the boolean AND between self and the other edge collection in-place (modifying self)
	Edges	and_with	(const Region other)	Selects the parts of the edges inside the given region in-place (modifying self)
[const]	Edges[]	andnot	(const Edges other)	Returns the boolean AND and NOT between self and the other edge set
[const]	Edges[]	andnot	(const Region other)	Returns the boolean AND and NOT between self and the region
	void	assign	(const Edges other)	Assigns another object to self
[const]	Box	bbox		Returns the bounding box of the edge collection
[const]	Edges	centers	(unsigned int length, double fraction)	Returns edges representing the center part of the edges
	void	clear		Clears the edge collection
[const]	unsigned long	count		Returns the (flat) number of edges in the edge collection
[const]	unsigned long	data_id		Returns the data ID (a unique identifier for the underlying data storage)
	void	disable_progress		Disable progress reporting

[const]	new Edges ptr	**dup**		Creates a copy of self
[const,iter]	Edge	**each**		Returns each edge of the region
[const,iter]	Edge	**each_merged**		Returns each edge of the region
	void	**enable_progress**	(string label)	Enable progress reporting
	void	**enable_properties**		Enables properties for the given container.
[const]	EdgePairs	**enclosed_check**	(const Edges other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)	Performs an inside check with options
[const]	EdgePairs	**enclosing_check**	(const Edges other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTo	Performs an enclosing check with options
[const]	Edges	**end_segments**	(unsigned int length, double fraction)	Returns edges representing a part of the edge before the end point
[const]	Region	**extended**	(int b, int e, int o, int i, bool join)	Returns a region with shapes representing the edges with the specified extensions
[const]	Region	**extended_in**	(int e)	Returns a region with shapes representing the edges with the given width
[const]	Region	**extended_out**	(int e)	Returns a region with shapes representing the edges with the given width
[const]	Region	**extents**		Returns a region with the bounding boxes of the edges
[const]	Region	**extents**	(int d)	Returns a region with the enlarged bounding boxes of the edges

[const]	Region	**extents**	(int dx, int dy)	Returns a region with the enlarged bounding boxes of the edges
	void	**filter**	(const EdgeFilter ptr filter)	Applies a generic filter in place (replacing the edges from the Edges collection)
	void	**filter_properties**	(variant[] keys)	Filters properties by certain keys.
[const]	Edges	**filtered**	(const EdgeFilter ptr filtered)	Applies a generic filter and returns a filtered copy
	void	**flatten**		Explicitly flattens an edge collection
[const]	bool	**has_valid_edges?**		Returns true if the edge collection is flat and individual edges can be accessed randomly
[const]	unsigned long	**hier_count**		Returns the (hierarchical) number of edges in the edge collection
[const]	Edges	**in**	(const Edges other)	Returns all edges which are members of the other edge collection
[const]	Edges[]	**in_and_out**	(const Edges other)	Returns all polygons which are members and not members of the other region
	void	**insert**	(const Edge edge)	Inserts an edge
	void	**insert**	(const Box box)	Inserts a box
	void	**insert**	(const Polygon polygon)	Inserts a polygon
	void	**insert**	(const SimplePolygon polygon)	Inserts a simple polygon
	void	**insert**	(const Path path)	Inserts a path
	void	**insert**	(const Edges edges)	Inserts all edges from the other edge collection into this one
	void	**insert**	(const Region region)	Inserts a region
	void	**insert**	(const Shapes shapes)	Inserts all edges from the shape collection into this edge collection
	void	**insert**	(const Shapes shapes, const Trans trans)	Inserts all edges from the shape collection into this edge collection (with transformation)
	void	**insert**	(const Shapes shapes, const ICplxTrans trans)	Inserts all edges from the shape collection into this edge collection with complex transformation
	void	**insert**	(RecursiveShapeIterator shape_iterator)	Inserts all shapes delivered by the recursive shape iterator into this edge collection
	void	**insert**	(RecursiveShapeIterator shape_iterator, ICplxTrans trans)	Inserts all shapes delivered by the recursive shape iterator into this edge collection with a transformation

	void	**insert**	(Polygon[] polygons)	Inserts all polygons from the array into this edge collection
	void	**insert**	(Edge[] edges)	Inserts all edges from the array into this edge collection
[const]	void	**insert_into**	(Layout ptr layout, unsigned int cell_index, unsigned int layer)	Inserts this edge collection into the given layout, below the given cell and into the given layer.
[const]	Edges	**inside**	(const Edges other)	Returns the edges of this edge collection which are inside (completely covered by) edges from the other edge collection
[const]	Edges	**inside**	(const Region other)	Returns the edges from this edge collection which are inside (completely covered by) polygons from the region
[const]	EdgePairs	**inside_check**	(const Edges other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)	Performs an inside check with options
[const]	Edges[]	**inside_outside_pa**	(const Region other)	Returns the partial edges inside and outside the given region
[const]	Edges	**inside_part**	(const Region other)	Returns the parts of the edges of this edge collection which are inside the polygons of the region
[const]	Edges	**interacting**	(const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the edges of this edge collection which overlap or touch edges from the other edge collection
[const]	Edges	**interacting**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the edges from this edge collection which overlap or touch polygons from the region
[const]	Edges	**intersections**	(const Edges other)	Computes the intersections between this edges and other edges
[const]	bool	**is_deep?**		Returns true if the edge collection is a deep (hierarchical) one
[const]	bool	**is_empty?**		Returns true if the edge collection is empty

[const]	bool	**is_merged?**		Returns true if the edge collection is merged
[const]	Edges	**join**	(const Edges other)	Returns the combined edge set of self and the other one
	Edges	**join_with**	(const Edges other)	Adds the edges of the other edge collection to self
[const]	unsigned int	**length**		Returns the total length of all edges in the edge collection
[const]	unsigned int	**length**	(const Box rect)	Returns the total length of all edges in the edge collection (restricted to a rectangle)
	void	**map_properties**	(map<variant,variant> key_map)	Maps properties by name key.
[const]	Edges	**members_of**	(const Edges other)	Returns all edges which are members of the other edge collection
	Edges	**merge**		Merge the edges
[const]	Edges	**merged**		Returns the merged edge collection
	void	**merged_semantics**	(bool f)	Enable or disable merged semantics
[const]	bool	**merged_semantics?**		Gets a flag indicating whether merged semantics is enabled
	Edges	**move**	(const Vector v)	Moves the edge collection
	Edges	**move**	(int x, int y)	Moves the edge collection
[const]	Edges	**moved**	(const Vector v)	Returns the moved edge collection (does not modify self)
[const]	Edges	**moved**	(int x, int v)	Returns the moved edge collection (does not modify self)
[const]	Edges	**not**	(const Edges other)	Returns the boolean NOT between self and the other edge collection
[const]	Edges	**not**	(const Region other)	Returns the parts of the edges outside the given region
[const]	Edges	**not_in**	(const Edges other)	Returns all edges which are not members of the other edge collection
[const]	Edges	**not_inside**	(const Edges other)	Returns the edges of this edge collection which are not inside (completely covered by) edges from the other edge collection
[const]	Edges	**not_inside**	(const Region other)	Returns the edges from this edge collection which are not inside (completely covered by) polygons from the region

[const]	Edges	**not_interacting**	(const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the edges of this edge collection which do not overlap or touch edges from the other edge collection
[const]	Edges	**not_interacting**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the edges from this edge collection which do not overlap or touch polygons from the region
[const]	Edges	**not_members_of**	(const Edges other)	Returns all edges which are not members of the other edge collection
[const]	Edges	**not_outside**	(const Edges other)	Returns the edges of this edge collection which are not outside (completely covered by) edges from the other edge collection
[const]	Edges	**not_outside**	(const Region other)	Returns the edges from this edge collection which are not outside (completely covered by) polygons from the region
	Edges	**not_with**	(const Edges other)	Performs the boolean NOT between self and the other edge collection in-place (modifying self)
	Edges	**not_with**	(const Region other)	Selects the parts of the edges outside the given region in-place (modifying self)
[const]	Edges	**or**	(const Edges other)	Returns the boolean OR between self and the other edge set
	Edges	**or_with**	(const Edges other)	Performs the boolean OR between self and the other edge set in-place (modifying self)
[const]	Edges	**outside**	(const Edges other)	Returns the edges of this edge collection which are outside (completely covered by) edges from the other edge collection
[const]	Edges	**outside**	(const Region other)	Returns the edges from this edge collection which are outside (completely covered by) polygons from the region
[const]	Edges	**outside_part**	(const Region other)	Returns the parts of the edges of this edge collection which are outside the polygons of the region
[const]	EdgePairs	**overlap_check**	(const Edges other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max,	Performs an overlap check with options

			ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)	
	void	**process**	(const EdgeOperator ptr process)	Applies a generic edge processor in place (replacing the edges from the Edges collection)
[const]	Edges	**processed**	(const EdgeOperator ptr processed)	Applies a generic edge processor and returns a processed copy
[const]	EdgePairs	**processed**	(const EdgeToEdgePairOperator ptr processed)	Applies a generic edge-to-edge-pair processor and returns an edge pair collection with the results
[const]	Region	**processed**	(const EdgeToPolygonOperator ptr processed)	Applies a generic edge-to-polygon processor and returns an edge collection with the results
[const]	Region	**pull_interacting**	(const Region other)	Returns all polygons of "other" which are interacting with (overlapping, touching) edges of this edge set
[const]	Edges	**pull_interacting**	(const Edges other)	Returns all edges of "other" which are interacting with polygons of this edge set
	void	**remove_properties**		Removes properties for the given container.
	Edges	**select_inside**	(const Edges other)	Selects the edges from this edge collection which are inside (completely covered by) edges from the other edge collection
	Edges	**select_inside**	(const Region other)	Selects the edges from this edge collection which are inside (completely covered by) polygons from the region
	Edges	**select_inside_part**	(const Region other)	Selects the parts of the edges from this edge collection which are inside the polygons of the given region
	Edges	**select_interacting**	(const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the edges from this edge collection which overlap or touch edges from the other edge collection
	Edges	**select_interacting**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the edges from this edge collection which overlap or touch polygons from the region
	Edges	**select_not_inside**	(const Edges other)	Selects the edges from this edge collection which are not inside (completely covered by) edges from the other edge collection
	Edges	**select_not_inside**	(const Region other)	Selects the edges from this edge collection which are not inside (completely covered by) polygons from the region

	Edges	select_not_interac	(const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the edges from this edge collection which do not overlap or touch edges from the other edge collection
	Edges	select_not_interacting	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the edges from this edge collection which do not overlap or touch polygons from the region
	Edges	select_not_outside	(const Edges other)	Selects the edges from this edge collection which are not outside (completely covered by) edges from the other edge collection
	Edges	select_not_outside	(const Region other)	Selects the edges from this edge collection which are not outside (completely covered by) polygons from the region
	Edges	select_outside	(const Edges other)	Selects the edges from this edge collection which are outside (completely covered by) edges from the other edge collection
	Edges	select_outside	(const Region other)	Selects the edges from this edge collection which are outside (completely covered by) polygons from the region
	Edges	select_outside_pa	(const Region other)	Selects the parts of the edges from this edge collection which are outside the polygons of the given region
[const]	EdgePairs	separation_check	(const Edges other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)	Performs an overlap check with options
[const]	EdgePairs	space_check	(int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTo	Performs a space check with options

[const]	Edges[]	**split_inside**	(const Edges other)	Selects the edges from this edge collection which are and are not inside (completely covered by) edges from the other collection
[const]	Edges[]	**split_inside**	(const Region other)	Selects the edges from this edge collection which are and are not inside (completely covered by) polygons from the other region
[const]	Edges[]	**split_interacting**	(const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the edges from this edge collection which do and do not interact with edges from the other collection
[const]	Edges[]	**split_interacting**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the edges from this edge collection which do and do not interact with polygons from the other region
[const]	Edges[]	**split_outside**	(const Edges other)	Selects the edges from this edge collection which are and are not outside (completely covered by) edges from the other collection
[const]	Edges[]	**split_outside**	(const Region other)	Selects the edges from this edge collection which are and are not outside (completely covered by) polygons from the other region
[const]	Edges	**start_segments**	(unsigned int length, double fraction)	Returns edges representing a part of the edge after the start point
	void	**swap**	(Edges other)	Swap the contents of this edge collection with the contents of another one
[const]	string	**to_s**		Converts the edge collection to a string
[const]	string	**to_s**	(unsigned long max_count)	Converts the edge collection to a string
	Edges	**transform**	(const Trans t)	Transform the edge collection (modifies self)
	Edges	**transform**	(const ICplxTrans t)	Transform the edge collection with a complex transformation (modifies self)
	Edges	**transform**	(const IMatrix2d t)	Transform the edge collection (modifies self)
	Edges	**transform**	(const IMatrix3d t)	Transform the edge collection (modifies self)
[const]	Edges	**transformed**	(const Trans t)	Transform the edge collection
[const]	Edges	**transformed**	(const ICplxTrans t)	Transform the edge collection with a complex transformation
[const]	Edges	**transformed**	(const IMatrix2d t)	Transform the edge collection
[const]	Edges	**transformed**	(const IMatrix3d t)	Transform the edge collection
[const]	EdgePairs	**width_check**	(int d, bool whole_edges = false,	Performs a width check with options

			Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)	
[const]	Edges	**with_angle**	(double angle, bool inverse)	Filters the edges by orientation
[const]	Edges	**with_angle**	(double min_angle, double max_angle, bool inverse, bool include_min_angle = true, bool include_max_angle = false)	Filters the edges by orientation
[const]	Edges	**with_angle**	(Edges::EdgeType type, bool inverse)	Filters the edges by orientation type
[const]	Edges	**with_length**	(unsigned int length, bool inverse)	Filters the edges by length
[const]	Edges	**with_length**	(variant min_length, variant max_length, bool inverse)	Filters the edges by length
[const]	void	**write**	(string filename)	Writes the region to a file
[const]	Edges	**xor**	(const Edges other)	Returns the boolean XOR between self and the other edge collection
	Edges	**xor_with**	(const Edges other)	Performs the boolean XOR between self and the other edge collection in-place (modifying self)
[const]	Edges	**|**	(const Edges other)	Returns the boolean OR between self and the other edge set
	Edges	**|=**	(const Edges other)	Performs the boolean OR between self and the other edge set in-place (modifying self)

Public static methods and constants

[static,const]	Edges::EdgeType	**DiagonalEdges**	Diagonal edges are selected (-45 and 45 degree)
[static,const]	PropertyConstraint	**DifferentPropertiesConstraint**	Specifies to consider shapes only if their user properties are different
[static,const]	PropertyConstraint	**DifferentPropertiesConstra**	Specifies to consider shapes only if their user properties are different

[static,const]	Metrics	**Euclidian**	Specifies Euclidian metrics for the check functions
[static,const]	PropertyConstraint	**IgnoreProperties**	Specifies to ignore properties
[static,const]	ZeroDistanceMode	**IncludeZeroDistanceWhenCollinearAndTouching**	Specifies that check functions should include edges when they are collinear and touch
[static,const]	ZeroDistanceMode	**IncludeZeroDistanceWhen**	Specifies that check functions should include edges when they overlap
[static,const]	ZeroDistanceMode	**IncludeZeroDistanceWhenTouching**	Specifies that check functions should include edges when they touch
[static,const]	ZeroDistanceMode	**NeverIncludeZeroDistance**	Specifies that check functions should never include edges with zero distance.
[static,const]	PropertyConstraint	**NoPropertyConstraint**	Specifies not to apply any property constraint
[static,const]	Edges::EdgeType	**OrthoDiagonalEdges**	Diagonal or orthogonal edges are selected (0, 90, -45 and 45 degree)
[static,const]	Edges::EdgeType	**OrthoEdges**	Horizontal and vertical edges are selected
[static,const]	Metrics	**Projection**	Specifies projected distance metrics for the check functions
[static,const]	PropertyConstraint	**SamePropertiesConstraint**	Specifies to consider shapes only if their user properties are the same
[static,const]	PropertyConstraint	**SamePropertiesConstraint**	Specifies to consider shapes only if their user properties are the same
[static,const]	Metrics	**Square**	Specifies square metrics for the check functions

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	unsigned long	**size**		Use of this method is deprecated. Use count instead
	Edges	**transform_icplx**	(const ICplxTrans t)	Use of this method is deprecated. Use transform instead

[const]	Edges	**transformed_icplx**	(const ICplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

&

(1) Signature: *[const]* Edges **&** (const Edges other)

Description: Returns the boolean AND between self and the other edge collection

Returns: The result of the boolean AND operation

The boolean AND operation will return all parts of the edges in this collection which are coincident with parts of the edges in the other collection.The result will be a merged edge collection.

The 'and' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'and_' in Python.

(2) Signature: *[const]* Edges **&** (const Region other)

Description: Returns the parts of the edges inside the given region

Returns: The edges inside the given region

This operation returns the parts of the edges which are inside the given region. Edges on the borders of the polygons are included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

This method has been introduced in version 0.24.The 'and' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'and_' in Python.

&=

(1) Signature: Edges **&=** (const Edges other)

Description: Performs the boolean AND between self and the other edge collection in-place (modifying self)

Returns: The edge collection after modification (self)

The boolean AND operation will return all parts of the edges in this collection which are coincident with parts of the edges in the other collection.The result will be a merged edge collection.

Note that in Ruby, the '&=' operator actually does not exist, but is emulated by '&' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'and_with' instead.

The 'and_with' alias has been introduced in version 0.28.12.

(2) Signature: Edges **&=** (const Region other)

Description: Selects the parts of the edges inside the given region in-place (modifying self)

Returns: The edge collection after modification (self)

This operation selects the parts of the edges which are inside the given region. Edges on the borders of the polygons are included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

This method has been introduced in version 0.24. Note that in Ruby, the '&=' operator actually does not exist, but is emulated by '&' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'and_with' instead.

The 'and_with' alias has been introduced in version 0.28.12.

+

Signature: *[const]* Edges **+** (const Edges other)

Description: Returns the combined edge set of self and the other one

Returns: The resulting edge set

This operator adds the edges of the other edge set to self and returns a new combined edge set. This usually creates unmerged edge sets and edges may overlap. Use merge if you want to ensure the result edge set is merged.

The 'join' alias has been introduced in version 0.28.12.

+=

Signature: Edges **+=** (const Edges other)

Description: Adds the edges of the other edge collection to self

Returns: The edge set after modification (self)

This operator adds the edges of the other edge set to self. This usually creates unmerged edge sets and edges may overlap. Use merge if you want to ensure the result edge set is merged.

Note that in Ruby, the '+=' operator actually does not exist, but is emulated by '+' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'join_with' instead.

The 'join_with' alias has been introduced in version 0.28.12.

-

(1) Signature: *[const]* Edges **-** (const Edges other)

Description: Returns the boolean NOT between self and the other edge collection

Returns: The result of the boolean NOT operation

The boolean NOT operation will return all parts of the edges in this collection which are not coincident with parts of the edges in the other collection.The result will be a merged edge collection.

The 'not' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'not_' in Python.

(2) Signature: *[const]* Edges **-** (const Region other)

Description: Returns the parts of the edges outside the given region

Returns: The edges outside the given region

This operation returns the parts of the edges which are outside the given region. Edges on the borders of the polygons are not included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

This method has been introduced in version 0.24.The 'not' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'not_' in Python.

-=

(1) Signature: Edges **-=** (const Edges other)

Description: Performs the boolean NOT between self and the other edge collection in-place (modifying self)

Returns: The edge collection after modification (self)

The boolean NOT operation will return all parts of the edges in this collection which are not coincident with parts of the edges in the other collection.The result will be a merged edge collection.

Note that in Ruby, the '-=' operator actually does not exist, but is emulated by '-' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'not_with' instead.

The 'not_with' alias has been introduced in version 0.28.12.

(2) Signature: Edges **-=** (const Region other)

Description: Selects the parts of the edges outside the given region in-place (modifying self)

Returns: The edge collection after modification (self)

This operation selects the parts of the edges which are outside the given region. Edges on the borders of the polygons are not included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

Note that in Ruby, the '-=' operator actually does not exist, but is emulated by '-' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'not_with' instead.

This method has been introduced in version 0.24.The 'not_with' alias has been introduced in version 0.28.12.

DiagonalEdges

Signature: *[static,const]* Edges::EdgeType **DiagonalEdges**

Description: Diagonal edges are selected (-45 and 45 degree)

Python specific notes:
The object exposes a readable attribute 'DiagonalEdges'. This is the getter.

DifferentPropertiesConstraint

Signature: *[static,const]* PropertyConstraint **DifferentPropertiesConstraint**

Description: Specifies to consider shapes only if their user properties are different

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are different. Properties are generated on the output shapes where applicable.

Python specific notes:
The object exposes a readable attribute 'DifferentPropertiesConstraint'. This is the getter.

DifferentPropertiesConstraintDrop

Signature: *[static,const]* PropertyConstraint **DifferentPropertiesConstraintDrop**

Description: Specifies to consider shapes only if their user properties are different

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are the same. No properties are generated on the output shapes.

Python specific notes:
The object exposes a readable attribute 'DifferentPropertiesConstraintDrop'. This is the getter.

Euclidian

Signature: *[static,const]* Metrics **Euclidian**

Description: Specifies Euclidian metrics for the check functions

This value can be used for the metrics parameter in the check functions, i.e. width_check. This value specifies Euclidian metrics, i.e. the distance between two points is measured by:

```
d = sqrt(dx^2 + dy^2)
```

All points within a circle with radius d around one point are considered to have a smaller distance than d.

Python specific notes:
The object exposes a readable attribute 'Euclidian'. This is the getter.

IgnoreProperties

Signature: *[static,const]* PropertyConstraint **IgnoreProperties**

Description: Specifies to ignore properties

When using this constraint - for example on a boolean operation - properties are ignored and are not generated in the output.

Python specific notes:
The object exposes a readable attribute 'IgnoreProperties'. This is the getter.

IncludeZeroDistanceWhenCollinearAndTouching

Signature: *[static,const]* ZeroDistanceMode **IncludeZeroDistanceWhenCollinearAndTouching**

Description: Specifies that check functions should include edges when they are collinear and touch

With this specification, the check functions will also check edges if they share at least one common point and are collinear. This is the mode that includes checking the 'kissing corner' cases when the kissing edges are collinear. This mode was default up to version 0.28.

Python specific notes:
The object exposes a readable attribute 'IncludeZeroDistanceWhenCollinearAndTouching'. This is the getter.

IncludeZeroDistanceWhe

Signature: *[static,const]* ZeroDistanceMode **IncludeZeroDistanceWhenOverlapping**

Description: Specifies that check functions should include edges when they overlap

With this specification, the check functions will also check edges which are collinear and share more than a single point. This is the mode that excludes the 'kissing corner' cases.

Python specific notes:
The object exposes a readable attribute 'IncludeZeroDistanceWhenOverlapping'. This is the getter.

IncludeZeroDistanceWhenTouching

Signature: *[static,const]* ZeroDistanceMode **IncludeZeroDistanceWhenTouching**

Description: Specifies that check functions should include edges when they touch

With this specification, the check functions will also check edges if they share at least one common point. This is the mode that includes checking the 'kissing corner' cases. This mode is default for version 0.28.16 and later.

Python specific notes:
The object exposes a readable attribute 'IncludeZeroDistanceWhenTouching'. This is the getter.

NeverIncludeZeroDistanc

Signature: *[static,const]* ZeroDistanceMode **NeverIncludeZeroDistance**

Description: Specifies that check functions should never include edges with zero distance.

With this specification, the check functions will ignore edges which are collinear or touch.

Python specific notes:
The object exposes a readable attribute 'NeverIncludeZeroDistance'. This is the getter.

NoPropertyConstraint

Signature: *[static,const]* PropertyConstraint **NoPropertyConstraint**

Description: Specifies not to apply any property constraint

When using this constraint - for example on a boolean operation - shapes are considered regardless of their user properties. Properties are generated on the output shapes where applicable.

Python specific notes:
The object exposes a readable attribute 'NoPropertyConstraint'. This is the getter.

OrthoDiagonalEdges

Signature: *[static,const]* Edges::EdgeType **OrthoDiagonalEdges**

Description: Diagonal or orthogonal edges are selected (0, 90, -45 and 45 degree)

Python specific notes:

The object exposes a readable attribute 'OrthoDiagonalEdges'. This is the getter.

OrthoEdges

Signature: *[static,const]* Edges::EdgeType **OrthoEdges**

Description: Horizontal and vertical edges are selected

Python specific notes:
The object exposes a readable attribute 'OrthoEdges'. This is the getter.

Projection

Signature: *[static,const]* Metrics **Projection**

Description: Specifies projected distance metrics for the check functions

This value can be used for the metrics parameter in the check functions, i.e. width_check. This value specifies projected metrics, i.e. the distance is defined as the minimum distance measured perpendicular to one edge. That implies that the distance is defined only where two edges have a non-vanishing projection onto each other.

Python specific notes:
The object exposes a readable attribute 'Projection'. This is the getter.

SamePropertiesConstraint

Signature: *[static,const]* PropertyConstraint **SamePropertiesConstraint**

Description: Specifies to consider shapes only if their user properties are the same

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are the same. Properties are generated on the output shapes where applicable.

Python specific notes:
The object exposes a readable attribute 'SamePropertiesConstraint'. This is the getter.

SamePropertiesConstrain

Signature: *[static,const]* PropertyConstraint **SamePropertiesConstraintDrop**

Description: Specifies to consider shapes only if their user properties are the same

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are the same. No properties are generated on the output shapes.

Python specific notes:
The object exposes a readable attribute 'SamePropertiesConstraintDrop'. This is the getter.

Square

Signature: *[static,const]* Metrics **Square**

Description: Specifies square metrics for the check functions

This value can be used for the metrics parameter in the check functions, i.e. width_check. This value specifies square metrics, i.e. the distance between two points is measured by:

```
d = max(abs(dx), abs(dy))
```

All points within a square with length 2*d around one point are considered to have a smaller distance than d in this metrics.

Python specific notes:
The object exposes a readable attribute 'Square'. This is the getter.

[]

Signature: *[const]* const Edge ptr **[]** (unsigned long n)

Description: Returns the nth edge of the collection

This method returns nil if the index is out of range. It is available for flat edge collections only - i.e. those for which has_valid_edges? is true. Use flatten to explicitly flatten an edge collection. This method returns the raw edge (not merged edges, even if merged semantics is enabled).

The each iterator is the more general approach to access the edges.

^

Signature: *[const]* Edges **^** (const Edges other)

Description: Returns the boolean XOR between self and the other edge collection

Returns: The result of the boolean XOR operation

The boolean XOR operation will return all parts of the edges in this and the other collection except the parts where both are coincident. The result will be a merged edge collection.

The 'xor' alias has been introduced in version 0.28.12.

^=

Signature: Edges **^=** (const Edges other)

Description: Performs the boolean XOR between self and the other edge collection in-place (modifying self)

Returns: The edge collection after modification (self)

The boolean XOR operation will return all parts of the edges in this and the other collection except the parts where both are coincident. The result will be a merged edge collection.

Note that in Ruby, the '^=' operator actually does not exist, but is emulated by '^' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'xor_with' instead.

The 'xor_with' alias has been introduced in version 0.28.12.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known

not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

and

(1) Signature: *[const]* Edges **and** (const Edges other)

Description: Returns the boolean AND between self and the other edge collection

Returns: The result of the boolean AND operation

The boolean AND operation will return all parts of the edges in this collection which are coincident with parts of the edges in the other collection.The result will be a merged edge collection.

The 'and' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'and_' in Python.

(2) Signature: *[const]* Edges **and** (const Region other)

Description: Returns the parts of the edges inside the given region

Returns: The edges inside the given region

This operation returns the parts of the edges which are inside the given region. Edges on the borders of the polygons are included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

This method has been introduced in version 0.24.The 'and' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'and_' in Python.

and_with

(1) Signature: Edges **and_with** (const Edges other)

Description: Performs the boolean AND between self and the other edge collection in-place (modifying self)

Returns: The edge collection after modification (self)

The boolean AND operation will return all parts of the edges in this collection which are coincident with parts of the edges in the other collection.The result will be a merged edge collection.

Note that in Ruby, the '&=' operator actually does not exist, but is emulated by '&' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'and_with' instead.

The 'and_with' alias has been introduced in version 0.28.12.

(2) Signature: Edges **and_with** (const Region other)

Description: Selects the parts of the edges inside the given region in-place (modifying self)

Returns: The edge collection after modification (self)

This operation selects the parts of the edges which are inside the given region. Edges on the borders of the polygons are included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

This method has been introduced in version 0.24. Note that in Ruby, the '&=' operator actually does not exist, but is emulated by '&' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'and_with' instead.

The 'and_with' alias has been introduced in version 0.28.12.

andnot

(1) Signature: *[const]* Edges[] **andnot** (const Edges other)

Description: Returns the boolean AND and NOT between self and the other edge set

Returns: A two-element array of edge collections with the first one being the AND result and the second one being the NOT result

This method will compute the boolean AND and NOT between two edge sets simultaneously. Because this requires a single sweep only, using this method is faster than doing AND and NOT separately.

This method has been added in version 0.28.

(2) Signature: *[const]* Edges[] **andnot** (const Region other)

Description: Returns the boolean AND and NOT between self and the region

Returns: A two-element array of edge collections with the first one being the AND result and the second one being the NOT result

This method will compute the boolean AND and NOT simultaneously. Because this requires a single sweep only, using this method is faster than doing AND and NOT separately.

This method has been added in version 0.28.

assign

Signature: void **assign** (const Edges other)

Description: Assigns another object to self

bbox

Signature: *[const]* Box **bbox**

Description: Returns the bounding box of the edge collection

The bounding box is the box enclosing all points of all edges.

centers

Signature: *[const]* Edges **centers** (unsigned int length, double fraction)

Description: Returns edges representing the center part of the edges

Returns: A new collection of edges representing the part around the center

This method allows one to specify the length of these segments in a twofold way: either as a fixed length or by specifying a fraction of the original length:

```
edges = ...  # An edge collection
edges.centers(100, 0.0)     # All segments have a length of 100 DBU
edges.centers(0, 50.0)      # All segments have a length of half the original
 length
edges.centers(100, 50.0)    # All segments have a length of half the original
 length
                            # or 100 DBU, whichever is larger
```

	It is possible to specify 0 for both values. In this case, degenerated edges (points) are delivered which specify the centers of the edges but can't participate in some functions.
clear	**Signature**: void **clear** **Description**: Clears the edge collection
count	**Signature**: *[const]* unsigned long **count** **Description**: Returns the (flat) number of edges in the edge collection This returns the number of raw edges (not merged edges if merged semantics is enabled). The count is computed 'as if flat', i.e. edges inside a cell are multiplied by the number of times a cell is instantiated. Starting with version 0.27, the method is called 'count' for consistency with Region. 'size' is still provided as an alias. **Python specific notes:** This method is also available as 'len(object)'.
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
data_id	**Signature**: *[const]* unsigned long **data_id** **Description**: Returns the data ID (a unique identifier for the underlying data storage) This method has been added in version 0.26.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
disable_progress	**Signature**: void **disable_progress** **Description**: Disable progress reporting Calling this method will disable progress reporting. See enable_progress.
dup	**Signature**: *[const]* new Edges ptr **dup** **Description**: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each

Signature: *[const,iter]* Edge **each**

Description: Returns each edge of the region

Python specific notes:
This method enables iteration of the object.

each_merged

Signature: *[const,iter]* Edge **each_merged**

Description: Returns each edge of the region

In contrast to each, this method delivers merged edges if merge semantics applies while each delivers the original edges only.

This method has been introduced in version 0.25.

enable_progress

Signature: void **enable_progress** (string label)

Description: Enable progress reporting

After calling this method, the edge collection will report the progress through a progress bar while expensive operations are running. The label is a text which is put in front of the progress bar. Using a progress bar will imply a performance penalty of a few percent typically.

enable_properties

Signature: void **enable_properties**

Description: Enables properties for the given container.

This method has an effect mainly on original layers and will import properties from such layers. By default, properties are not enabled on original layers. Alternatively you can apply filter_properties or map_properties to enable properties with a specific name key.

This method has been introduced in version 0.28.4.

enclosed_check

Signature: *[const]* EdgePairs **enclosed_check** (const Edges other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs an inside check with options

d:	The minimum distance for which the edges are checked
other:	The other edge collection against which to check
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The threshold angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper threshold of the projected length of one edge onto another
zero_distance_mode:	Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants. Use nil for this value to select the default (Euclidian metrics).

"ignore_angle" specifies the angle threshold of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one threshold, pass nil to the respective value.

The 'enclosed_check' alias was introduced in version 0.27.5. 'zero_distance_mode' has been added in version 0.29.

enclosing_check

Signature: *[const]* EdgePairs **enclosing_check** (const Edges other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs an enclosing check with options

d:	The minimum distance for which the edges are checked
other:	The other edge collection against which to check
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The threshold angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper threshold of the projected length of one edge onto another
zero_distance_mode:	Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants. Use nil for this value to select the default (Euclidian metrics).

"ignore_angle" specifies the angle threshold of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one threshold, pass nil to the respective value.

'zero_distance_mode' has been added in version 0.29.

end_segments

Signature: *[const]* Edges **end_segments** (unsigned int length, double fraction)

Description: Returns edges representing a part of the edge before the end point

Returns:	A new collection of edges representing the end part

This method allows one to specify the length of these segments in a twofold way: either as a fixed length or by specifying a fraction of the original length:

```
edges = ...  # An edge collection
edges.end_segments(100, 0.0)     # All segments have a length of 100 DBU
edges.end_segments(0, 50.0)      # All segments have a length of half the
 original length
edges.end_segments(100, 50.0)    # All segments have a length of half the
 original length
```

```
# or 100 DBU, whichever is larger
```

It is possible to specify 0 for both values. In this case, degenerated edges (points) are delivered which specify the end positions of the edges but can't participate in some functions.

extended

Signature: *[const]* Region **extended** (int b, int e, int o, int i, bool join)

Description: Returns a region with shapes representing the edges with the specified extensions

b:	the parallel extension at the start point of the edge
e:	the parallel extension at the end point of the edge
o:	the perpendicular extension to the "outside" (left side as seen in the direction of the edge)
i:	the perpendicular extension to the "inside" (right side as seen in the direction of the edge)
join:	If true, connected edges are joined before the extension is applied
Returns:	A region containing the polygons representing these extended edges

This is a generic version of extended_in and extended_out. It allows one to specify extensions for all four directions of an edge and to join the edges before the extension is applied.

For degenerated edges forming a point, a rectangle with the b, e, o and i used as left, right, top and bottom distance to the center point of this edge is created.

If join is true and edges form a closed loop, the b and e parameters are ignored and a rim polygon is created that forms the loop with the outside and inside extension given by o and i.

extended_in

Signature: *[const]* Region **extended_in** (int e)

Description: Returns a region with shapes representing the edges with the given width

e:	The extension width
Returns:	A region containing the polygons representing these extended edges

The edges are extended to the "inside" by the given distance "e". The distance will be applied to the right side as seen in the direction of the edge. By definition, this is the side pointing to the inside of the polygon if the edge was derived from a polygon.

Other versions of this feature are extended_out and extended.

extended_out

Signature: *[const]* Region **extended_out** (int e)

Description: Returns a region with shapes representing the edges with the given width

e:	The extension width
Returns:	A region containing the polygons representing these extended edges

The edges are extended to the "outside" by the given distance "e". The distance will be applied to the left side as seen in the direction of the edge. By definition, this is the side pointing to the outside of the polygon if the edge was derived from a polygon.

Other versions of this feature are extended_in and extended.

extents

(1) Signature: *[const]* Region **extents**

Description: Returns a region with the bounding boxes of the edges

This method will return a region consisting of the bounding boxes of the edges. The boxes will not be merged, so it is possible to determine overlaps of these boxes for example.

(2) Signature: *[const]* Region **extents** (int d)

Description: Returns a region with the enlarged bounding boxes of the edges

This method will return a region consisting of the bounding boxes of the edges enlarged by the given distance d. The enlargement is specified per edge, i.e the width and height will be increased by 2*d. The boxes will not be merged, so it is possible to determine overlaps of these boxes for example.

(3) Signature: *[const]* Region **extents** (int dx, int dy)

Description: Returns a region with the enlarged bounding boxes of the edges

This method will return a region consisting of the bounding boxes of the edges enlarged by the given distance dx in x direction and dy in y direction. The enlargement is specified per edge, i.e the width will be increased by 2*dx. The boxes will not be merged, so it is possible to determine overlaps of these boxes for example.

filter

Signature: void **filter** (const EdgeFilter ptr filter)

Description: Applies a generic filter in place (replacing the edges from the Edges collection)

See EdgeFilter for a description of this feature.

This method has been introduced in version 0.29.

filter_properties

Signature: void **filter_properties** (variant[] keys)

Description: Filters properties by certain keys.

Calling this method on a container will reduce the properties to values with name keys from the 'keys' list. As a side effect, this method enables properties on original layers.

This method has been introduced in version 0.28.4.

filtered

Signature: *[const]* Edges **filtered** (const EdgeFilter ptr filtered)

Description: Applies a generic filter and returns a filtered copy

See EdgeFilter for a description of this feature.

This method has been introduced in version 0.29.

flatten

Signature: void **flatten**

Description: Explicitly flattens an edge collection

If the collection is already flat (i.e. has_valid_edges? returns true), this method will not change it.

This method has been introduced in version 0.26.

has_valid_edges?

Signature: *[const]* bool **has_valid_edges?**

Description: Returns true if the edge collection is flat and individual edges can be accessed randomly

This method has been introduced in version 0.26.

hier_count

Signature: *[const]* unsigned long **hier_count**

Description: Returns the (hierarchical) number of edges in the edge collection

This returns the number of raw edges (not merged edges if merged semantics is enabled). The count is computed 'hierarchical', i.e. edges inside a cell are counted once even if the cell is instantiated multiple times.

This method has been introduced in version 0.27.

in

Signature: *[const]* Edges **in** (const Edges other)

Description: Returns all edges which are members of the other edge collection

This method returns all edges in self which can be found in the other edge collection as well with exactly the same geometry.

Python specific notes:
This attribute is available as 'in_' in Python.

in_and_out

Signature: *[const]* Edges[] **in_and_out** (const Edges other)

Description: Returns all polygons which are members and not members of the other region

This method is equivalent to calling members_of and not_members_of, but delivers both results at the same time and is more efficient than two separate calls. The first element returned is the members_of part, the second is the not_members_of part.

This method has been introduced in version 0.28.

insert

(1) Signature: void **insert** (const Edge edge)

Description: Inserts an edge

Inserts the edge into the edge collection.

(2) Signature: void **insert** (const Box box)

Description: Inserts a box

Inserts the edges that form the contour of the box into the edge collection.

(3) Signature: void **insert** (const Polygon polygon)

Description: Inserts a polygon

Inserts the edges that form the contour of the polygon into the edge collection.

(4) Signature: void **insert** (const SimplePolygon polygon)

Description: Inserts a simple polygon

Inserts the edges that form the contour of the simple polygon into the edge collection.

(5) Signature: void **insert** (const Path path)

Description: Inserts a path

Inserts the edges that form the contour of the path into the edge collection.

(6) Signature: void **insert** (const Edges edges)

Description: Inserts all edges from the other edge collection into this one

This method has been introduced in version 0.25.

(7) Signature: void **insert** (const Region region)

Description: Inserts a region

Inserts the edges that form the contours of the polygons from the region into the edge collection.

This method has been introduced in version 0.25.

(8) Signature: void **insert** (const Shapes shapes)

Description: Inserts all edges from the shape collection into this edge collection

This method takes each edge from the shape collection and inserts it into the region. "Polygon-like" objects are inserted as edges forming the contours of the polygons. Text objects are ignored.

This method has been introduced in version 0.25.

(9) Signature: void **insert** (const Shapes shapes, const Trans trans)

Description: Inserts all edges from the shape collection into this edge collection (with transformation)

This method acts as the version without transformation, but will apply the given transformation before inserting the edges.

This method has been introduced in version 0.25.

(10) Signature: void **insert** (const Shapes shapes, const ICplxTrans trans)

Description: Inserts all edges from the shape collection into this edge collection with complex transformation

This method acts as the version without transformation, but will apply the given complex transformation before inserting the edges.

This method has been introduced in version 0.25.

(11) Signature: void **insert** (RecursiveShapeIterator shape_iterator)

Description: Inserts all shapes delivered by the recursive shape iterator into this edge collection

For "solid" shapes (boxes, polygons, paths), this method inserts the edges that form the contour of the shape into the edge collection. Edge shapes are inserted as such. Text objects are not inserted, because they cannot be converted to polygons.

(12) Signature: void **insert** (RecursiveShapeIterator shape_iterator, ICplxTrans trans)

Description: Inserts all shapes delivered by the recursive shape iterator into this edge collection with a transformation

For "solid" shapes (boxes, polygons, paths), this method inserts the edges that form the contour of the shape into the edge collection. Edge shapes are inserted as such. Text objects are not inserted, because they cannot be converted to polygons. This variant will apply the given transformation to the shapes. This is useful to scale the shapes to a specific database unit for example.

(13) Signature: void **insert** (Polygon[] polygons)

Description: Inserts all polygons from the array into this edge collection

(14) Signature: void **insert** (Edge[] edges)

Description: Inserts all edges from the array into this edge collection

insert_into

Signature: *[const]* void **insert_into** (Layout ptr layout, unsigned int cell_index, unsigned int layer)

Description: Inserts this edge collection into the given layout, below the given cell and into the given layer.

If the edge collection is a hierarchical one, a suitable hierarchy will be built below the top cell or and existing hierarchy will be reused.

This method has been introduced in version 0.26.

inside

(1) Signature: *[const]* Edges **inside** (const Edges other)

Description: Returns the edges of this edge collection which are inside (completely covered by) edges from the other edge collection

Returns: A new edge collection containing the edges overlapping or touching edges from the other edge collection

This method has been introduced in version 0.28.

(2) Signature: *[const]* Edges **inside** (const Region other)

Description: Returns the edges from this edge collection which are inside (completely covered by) polygons from the region

Returns: A new edge collection containing the edges overlapping or touching polygons from the region

This method has been introduced in version 0.28.

inside_check

Signature: *[const]* EdgePairs **inside_check** (const Edges other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs an inside check with options

d: The minimum distance for which the edges are checked

other: The other edge collection against which to check

whole_edges: If true, deliver the whole edges

metrics: Specify the metrics type

ignore_angle: The threshold angle above which no check is performed

min_projection: The lower threshold of the projected length of one edge onto another

max_projection: The upper threshold of the projected length of one edge onto another

zero_distance_mode: Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants. Use nil for this value to select the default (Euclidian metrics).

"ignore_angle" specifies the angle threshold of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one threshold, pass nil to the respective value.

The 'enclosed_check' alias was introduced in version 0.27.5. 'zero_distance_mode' has been added in version 0.29.

inside_outside_part

Signature: *[const]* Edges[] **inside_outside_part** (const Region other)

Description: Returns the partial edges inside and outside the given region

Returns: A two-element array of edge collections with the first one being the inside_part result and the second one being the outside_part result

This method will compute the results simultaneously. Because this requires a single sweep only, using this method is faster than doing inside_part and outside_part separately.

This method has been added in version 0.28.

inside_part

Signature: *[const]* Edges **inside_part** (const Region other)

Description: Returns the parts of the edges of this edge collection which are inside the polygons of the region

Returns: A new edge collection containing the edge parts inside the region

This operation returns the parts of the edges which are inside the given region. This functionality is similar to the '&' operator, but edges on the borders of the polygons are not included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

This method has been introduced in version 0.24.

interacting

(1) Signature: *[const]* Edges **interacting** (const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the edges of this edge collection which overlap or touch edges from the other edge collection

Returns: A new edge collection containing the edges overlapping or touching edges from the other edge collection

'min_count' and 'max_count' impose a constraint on the number of times an edge of this collection has to interact with (different) edges of the other collection to make the edge selected. An edge is selected by this method if the number of edges interacting with an edge of this collection is between min_count and max_count (including max_count).

'min_count' and 'max_count' have been introduced in version 0.29.

(2) Signature: *[const]* Edges **interacting** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the edges from this edge collection which overlap or touch polygons from the region

Returns: A new edge collection containing the edges overlapping or touching polygons from the region

'min_count' and 'max_count' impose a constraint on the number of times an edge of this collection has to interact with (different) polygons of the other region to make the edge selected. An edge is selected by this method if the number of polygons interacting with an edge of this collection is between min_count and max_count (including max_count).

'min_count' and 'max_count' have been introduced in version 0.29.

intersections

Signature: *[const]* Edges **intersections** (const Edges other)

Description: Computes the intersections between this edges and other edges

This computation is like an AND operation, but also including crossing points between non-coincident edges as degenerated (point-like) edges.

This method has been introduced in version 0.26.2

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_deep?

Signature: *[const]* bool **is_deep?**

Description: Returns true if the edge collection is a deep (hierarchical) one

This method has been added in version 0.26.

is_empty?

Signature: *[const]* bool **is_empty?**

Description: Returns true if the edge collection is empty

is_merged?

Signature: *[const]* bool **is_merged?**

Description: Returns true if the edge collection is merged

If the region is merged, coincident edges have been merged into single edges. You can ensure merged state by calling merge.

join

Signature: *[const]* Edges **join** (const Edges other)

Description: Returns the combined edge set of self and the other one

Returns: The resulting edge set

This operator adds the edges of the other edge set to self and returns a new combined edge set. This usually creates unmerged edge sets and edges may overlap. Use merge if you want to ensure the result edge set is merged.

The 'join' alias has been introduced in version 0.28.12.

join_with

Signature: Edges **join_with** (const Edges other)

Description: Adds the edges of the other edge collection to self

Returns: The edge set after modification (self)

This operator adds the edges of the other edge set to self. This usually creates unmerged edge sets and edges may overlap. Use merge if you want to ensure the result edge set is merged.

Note that in Ruby, the '+=' operator actually does not exist, but is emulated by '+' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'join_with' instead.

The 'join_with' alias has been introduced in version 0.28.12.

length

(1) Signature: *[const]* unsigned int **length**

Description: Returns the total length of all edges in the edge collection

Merged semantics applies for this method (see merged_semantics= of merged semantics)

(2) Signature: *[const]* unsigned int **length** (const Box rect)

Description: Returns the total length of all edges in the edge collection (restricted to a rectangle)

This version will compute the total length of all edges in the collection, restricting the computation to the given rectangle. Edges along the border are handled in a special way: they are counted when they are oriented with their inside side toward the rectangle (in other words: outside edges must coincide with the rectangle's border in order to be counted).

Merged semantics applies for this method (see merged_semantics= of merged semantics)

map_properties

Signature: void **map_properties** (map<variant,variant> key_map)

Description: Maps properties by name key.

Calling this method on a container will reduce the properties to values with name keys from the 'keys' hash and renames the properties. Properties not listed in the key map will be removed. As a side effect, this method enables properties on original layers.

This method has been introduced in version 0.28.4.

members_of

Signature: *[const]* Edges **members_of** (const Edges other)

Description: Returns all edges which are members of the other edge collection

This method returns all edges in self which can be found in the other edge collection as well with exactly the same geometry.

Python specific notes:
This attribute is available as 'in_' in Python.

merge

Signature: Edges **merge**

Description: Merge the edges

Returns: The edge collection after the edges have been merged (self).

Merging joins parallel edges which overlap or touch. Crossing edges are not merged. If the edge collection is already merged, this method does nothing

merged

Signature: *[const]* Edges **merged**

Description: Returns the merged edge collection

Returns: The edge collection after the edges have been merged.

Merging joins parallel edges which overlap or touch. Crossing edges are not merged. In contrast to merge, this method does not modify the edge collection but returns a merged copy.

merged_semantics=

Signature: void **merged_semantics=** (bool f)

Description: Enable or disable merged semantics

If merged semantics is enabled (the default), colinear, connected or overlapping edges will be considered as single edges.

Python specific notes:
The object exposes a writable attribute 'merged_semantics'. This is the setter.

merged_semantics?

Signature: *[const]* bool **merged_semantics?**

Description: Gets a flag indicating whether merged semantics is enabled

See merged_semantics= for a description of this attribute.

Python specific notes:
The object exposes a readable attribute 'merged_semantics'. This is the getter.

move

(1) Signature: Edges **move** (const Vector v)

Description: Moves the edge collection

v: The distance to move the edge collection.

Returns: The moved edge collection (self).

Moves the polygon by the given offset and returns the moved edge collection. The edge collection is overwritten.

Starting with version 0.25 the displacement type is a vector.

(2) Signature: Edges **move** (int x, int y)

Description: Moves the edge collection

x:	The x distance to move the edge collection.
y:	The y distance to move the edge collection.
Returns:	The moved edge collection (self).

Moves the edge collection by the given offset and returns the moved edge collection. The edge collection is overwritten.

moved

(1) Signature: *[const]* Edges **moved** (const Vector v)

Description: Returns the moved edge collection (does not modify self)

v:	The distance to move the edge collection.
Returns:	The moved edge collection.

Moves the edge collection by the given offset and returns the moved edge collection. The edge collection is not modified.

Starting with version 0.25 the displacement type is a vector.

(2) Signature: *[const]* Edges **moved** (int x, int v)

Description: Returns the moved edge collection (does not modify self)

x:	The x distance to move the edge collection.
y:	The y distance to move the edge collection.
Returns:	The moved edge collection.

Moves the edge collection by the given offset and returns the moved edge collection. The edge collection is not modified.

new

(1) Signature: *[static]* new Edges ptr **new**

Description: Default constructor

This constructor creates an empty edge collection.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Edges ptr **new** (const Edge edge)

Description: Constructor from a single edge

This constructor creates an edge collection with a single edge.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Edges ptr **new** (Polygon[] array)

Description: Constructor from a polygon array

This constructor creates an edge collection from an array of polygons. The edges form the contours of the polygons.

Python specific notes:

This method is the default initializer of the object.

(4) Signature: *[static]* new Edges ptr **new** (Edge[] array)

Description: Constructor from an edge array

This constructor creates an edge collection from an array of edges.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new Edges ptr **new** (const Box box)

Description: Box constructor

This constructor creates an edge collection from a box. The edges form the contour of the box.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new Edges ptr **new** (const Polygon polygon)

Description: Polygon constructor

This constructor creates an edge collection from a polygon. The edges form the contour of the polygon.

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new Edges ptr **new** (const SimplePolygon polygon)

Description: Simple polygon constructor

This constructor creates an edge collection from a simple polygon. The edges form the contour of the polygon.

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new Edges ptr **new** (const Path path)

Description: Path constructor

This constructor creates an edge collection from a path. The edges form the contour of the path.

Python specific notes:
This method is the default initializer of the object.

(9) Signature: *[static]* new Edges ptr **new** (const Shapes shapes, bool as_edges = true)

Description: Constructor of a flat edge collection from a Shapes container

If 'as_edges' is true, the shapes from the container will be converted to edges (i.e. polygon contours to edges). Otherwise, only edges will be taken from the container.

This method has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

(10) Signature: *[static]* new Edges ptr **new** (const RecursiveShapeIterator shape_iterator, bool as_edges = true)

Description: Constructor of a flat edge collection from a hierarchical shape set

This constructor creates an edge collection from the shapes delivered by the given recursive shape iterator. It feeds the shapes from a hierarchy of cells into a flat edge set.

Text objects are not inserted, because they cannot be converted to edges. Edge objects are inserted as such. If "as_edges" is true, "solid" objects (boxes, polygons, paths) are converted to edges which form the hull of these objects. If "as_edges" is false, solid objects are ignored.

```
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
r = RBA::Edges::new(layout.begin_shapes(cell, layer), false)
```

Python specific notes:
This method is the default initializer of the object.

(11) Signature: *[static]* new Edges ptr **new** (const RecursiveShapeIterator shape_iterator, const ICplxTrans trans, bool as_edges = true)

Description: Constructor of a flat edge collection from a hierarchical shape set with a transformation

This constructor creates an edge collection from the shapes delivered by the given recursive shape iterator. It feeds the shapes from a hierarchy of cells into a flat edge set. The transformation is useful to scale to a specific database unit for example.

Text objects are not inserted, because they cannot be converted to edges. Edge objects are inserted as such. If "as_edges" is true, "solid" objects (boxes, polygons, paths) are converted to edges which form the hull of these objects. If "as_edges" is false, solid objects are ignored.

```
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
dbu    = 0.1 # the target database unit
r = RBA::Edges::new(layout.begin_shapes(cell, layer),
 RBA::ICplxTrans::new(layout.dbu / dbu))
```

Python specific notes:
This method is the default initializer of the object.

(12) Signature: *[static]* new Edges ptr **new** (const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, bool as_edges = true)

Description: Constructor of a hierarchical edge collection

This constructor creates an edge collection from the shapes delivered by the given recursive shape iterator. It feeds the shapes from a hierarchy of cells into the hierarchical edge set. The edges remain within their original hierarchy unless other operations require the edges to be moved in the hierarchy.

Text objects are not inserted, because they cannot be converted to edges. Edge objects are inserted as such. If "as_edges" is true, "solid" objects (boxes, polygons, paths) are converted to edges which form the hull of these objects. If "as_edges" is false, solid objects are ignored.

```
dss    = RBA::DeepShapeStore::new
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
r = RBA::Edges::new(layout.begin_shapes(cell, layer), dss, false)
```

Python specific notes:

This method is the default initializer of the object.

(13) Signature: *[static]* new Edges ptr **new** (const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, const ICplxTrans trans, bool as_edges = true)

Description: Constructor of a hierarchical edge collection with a transformation

This constructor creates an edge collection from the shapes delivered by the given recursive shape iterator. It feeds the shapes from a hierarchy of cells into the hierarchical edge set. The edges remain within their original hierarchy unless other operations require the edges to be moved in the hierarchy. The transformation is useful to scale to a specific database unit for example.

Text objects are not inserted, because they cannot be converted to edges. Edge objects are inserted as such. If "as_edges" is true, "solid" objects (boxes, polygons, paths) are converted to edges which form the hull of these objects. If "as_edges" is false, solid objects are ignored.

```
dss    = RBA::DeepShapeStore::new
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
dbu    = 0.1 # the target database unit
r = RBA::Edges::new(layout.begin_shapes(cell, layer), dss,
 RBA::ICplxTrans::new(layout.dbu / dbu), false)
```

Python specific notes:
This method is the default initializer of the object.

(14) Signature: *[static]* new Edges ptr **new** (const RecursiveShapeIterator shape_iterator, string expr, bool as_pattern = true)

Description: Constructor from a text set

shape_iterator:	The iterator from which to derive the texts
expr:	The selection string
as_pattern:	If true, the selection string is treated as a glob pattern. Otherwise the match is exact.

This special constructor will create dot-like edges from the text objects delivered by the shape iterator. Each text object will give a degenerated edge (a dot) that represents the text origin. Texts can be selected by their strings - either through a glob pattern or by exact comparison with the given string. The following options are available:

```
dots = RBA::Edges::new(iter, "*")           # all texts
dots = RBA::Edges::new(iter, "A*")          # all texts starting with an 'A'
dots = RBA::Edges::new(iter, "A*", false)   # all texts exactly matching 'A*'
```

This method has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

(15) Signature: *[static]* new Edges ptr **new** (const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, string expr, bool as_pattern = true)

Description: Constructor from a text set

shape_iterator:	The iterator from which to derive the texts

dss:	The DeepShapeStore object that acts as a heap for hierarchical operations.
expr:	The selection string
as_pattern:	If true, the selection string is treated as a glob pattern. Otherwise the match is exact.

This special constructor will create a deep edge set from the text objects delivered by the shape iterator. Each text object will give a degenerated edge (a dot) that represents the text origin. Texts can be selected by their strings - either through a glob pattern or by exact comparison with the given string. The following options are available:

```
region = RBA::Region::new(iter, dss, "*")           # all texts
region = RBA::Region::new(iter, dss, "A*")          # all texts starting with
 an 'A'
region = RBA::Region::new(iter, dss, "A*", false)   # all texts exactly
 matching 'A*'
```

This method has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

not

(1) Signature: *[const]* Edges **not** (const Edges other)

Description: Returns the boolean NOT between self and the other edge collection

Returns:	The result of the boolean NOT operation

The boolean NOT operation will return all parts of the edges in this collection which are not coincident with parts of the edges in the other collection.The result will be a merged edge collection.

The 'not' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'not_' in Python.

(2) Signature: *[const]* Edges **not** (const Region other)

Description: Returns the parts of the edges outside the given region

Returns:	The edges outside the given region

This operation returns the parts of the edges which are outside the given region. Edges on the borders of the polygons are not included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

This method has been introduced in version 0.24.The 'not' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'not_' in Python.

not_in

Signature: *[const]* Edges **not_in** (const Edges other)

Description: Returns all edges which are not members of the other edge collection

This method returns all edges in self which can not be found in the other edge collection with exactly the same geometry.

not_inside

(1) Signature: *[const]* Edges **not_inside** (const Edges other)

Description: Returns the edges of this edge collection which are not inside (completely covered by) edges from the other edge collection

Returns: A new edge collection containing the edges not overlapping or touching edges from the other edge collection

This method has been introduced in version 0.28.

(2) Signature: *[const]* Edges **not_inside** (const Region other)

Description: Returns the edges from this edge collection which are not inside (completely covered by) polygons from the region

Returns: A new edge collection containing the edges not overlapping or touching polygons from the region

This method has been introduced in version 0.28.

not_interacting

(1) Signature: *[const]* Edges **not_interacting** (const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the edges of this edge collection which do not overlap or touch edges from the other edge collection

Returns: A new edge collection containing the edges not overlapping or touching edges from the other edge collection

'min_count' and 'max_count' impose a constraint on the number of times an edge of this collection has to interact with (different) edges of the other collection to make the edge selected. An edge is not selected by this method if the number of edges interacting with an edge of this collection is between min_count and max_count (including max_count).

'min_count' and 'max_count' have been introduced in version 0.29.

(2) Signature: *[const]* Edges **not_interacting** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the edges from this edge collection which do not overlap or touch polygons from the region

Returns: A new edge collection containing the edges not overlapping or touching polygons from the region

'min_count' and 'max_count' impose a constraint on the number of times an edge of this collection has to interact with (different) polygons of the other region to make the edge selected. An edge is not selected by this method if the number of polygons interacting with an edge of this collection is between min_count and max_count (including max_count).

'min_count' and 'max_count' have been introduced in version 0.29.

not_members_of

Signature: *[const]* Edges **not_members_of** (const Edges other)

Description: Returns all edges which are not members of the other edge collection

This method returns all edges in self which can not be found in the other edge collection with exactly the same geometry.

not_outside

(1) Signature: *[const]* Edges **not_outside** (const Edges other)

Description: Returns the edges of this edge collection which are not outside (completely covered by) edges from the other edge collection

Returns: A new edge collection containing the edges not overlapping or touching edges from the other edge collection

This method has been introduced in version 0.28.

(2) Signature: *[const]* Edges **not_outside** (const Region other)

Description: Returns the edges from this edge collection which are not outside (completely covered by) polygons from the region

Returns: A new edge collection containing the edges not overlapping or touching polygons from the region

This method has been introduced in version 0.28.

not_with

(1) Signature: Edges **not_with** (const Edges other)

Description: Performs the boolean NOT between self and the other edge collection in-place (modifying self)

Returns: The edge collection after modification (self)

The boolean NOT operation will return all parts of the edges in this collection which are not coincident with parts of the edges in the other collection.The result will be a merged edge collection.

Note that in Ruby, the '-=' operator actually does not exist, but is emulated by '-' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'not_with' instead.

The 'not_with' alias has been introduced in version 0.28.12.

(2) Signature: Edges **not_with** (const Region other)

Description: Selects the parts of the edges outside the given region in-place (modifying self)

Returns: The edge collection after modification (self)

This operation selects the parts of the edges which are outside the given region. Edges on the borders of the polygons are not included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

Note that in Ruby, the '-=' operator actually does not exist, but is emulated by '-' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'not_with' instead.

This method has been introduced in version 0.24.The 'not_with' alias has been introduced in version 0.28.12.

or

Signature: *[const]* Edges **or** (const Edges other)

Description: Returns the boolean OR between self and the other edge set

Returns: The resulting edge collection

The boolean OR is implemented by merging the edges of both edge sets. To simply join the edge collections without merging, the + operator is more efficient. The 'or' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'or_' in Python.

or_with

Signature: Edges **or_with** (const Edges other)

Description: Performs the boolean OR between self and the other edge set in-place (modifying self)

Returns: The edge collection after modification (self)

The boolean OR is implemented by merging the edges of both edge sets. To simply join the edge collections without merging, the + operator is more efficient. Note that in Ruby, the '|=' operator actually does not exist, but is emulated by '|' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'or_with' instead.

The 'or_with' alias has been introduced in version 0.28.12.

outside

(1) Signature: *[const]* Edges **outside** (const Edges other)

Description: Returns the edges of this edge collection which are outside (completely covered by) edges from the other edge collection

Returns: A new edge collection containing the edges overlapping or touching edges from the other edge collection

This method has been introduced in version 0.28.

(2) Signature: *[const]* Edges **outside** (const Region other)

Description: Returns the edges from this edge collection which are outside (completely covered by) polygons from the region

Returns: A new edge collection containing the edges overlapping or touching polygons from the region

This method has been introduced in version 0.28.

outside_part

Signature: *[const]* Edges **outside_part** (const Region other)

Description: Returns the parts of the edges of this edge collection which are outside the polygons of the region

Returns: A new edge collection containing the edge parts outside the region

This operation returns the parts of the edges which are not inside the given region. This functionality is similar to the '-' operator, but edges on the borders of the polygons are included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

This method has been introduced in version 0.24.

overlap_check

Signature: *[const]* EdgePairs **overlap_check** (const Edges other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs an overlap check with options

d: The minimum distance for which the edges are checked

other: The other edge collection against which to check

whole_edges: If true, deliver the whole edges

metrics: Specify the metrics type

ignore_angle: The threshold angle above which no check is performed

min_projection: The lower threshold of the projected length of one edge onto another

max_projection: The upper threshold of the projected length of one edge onto another

zero_distance_mode: Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants. Use nil for this value to select the default (Euclidian metrics).

"ignore_angle" specifies the angle threshold of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one threshold, pass nil to the respective value.

'zero_distance_mode' has been added in version 0.29.

process

Signature: void **process** (const EdgeOperator ptr process)

Description: Applies a generic edge processor in place (replacing the edges from the Edges collection)

See EdgeProcessor for a description of this feature.

This method has been introduced in version 0.29.

processed

(1) Signature: *[const]* Edges **processed** (const EdgeOperator ptr processed)

Description: Applies a generic edge processor and returns a processed copy

See EdgeProcessor for a description of this feature.

This method has been introduced in version 0.29.

(2) Signature: *[const]* EdgePairs **processed** (const EdgeToEdgePairOperator ptr processed)

Description: Applies a generic edge-to-edge-pair processor and returns an edge pair collection with the results

See EdgeToEdgePairProcessor for a description of this feature.

This method has been introduced in version 0.29.

(3) Signature: *[const]* Region **processed** (const EdgeToPolygonOperator ptr processed)

Description: Applies a generic edge-to-polygon processor and returns an edge collection with the results

See EdgeToPolygonProcessor for a description of this feature.

This method has been introduced in version 0.29.

pull_interacting

(1) Signature: *[const]* Region **pull_interacting** (const Region other)

Description: Returns all polygons of "other" which are interacting with (overlapping, touching) edges of this edge set

Returns: The region after the polygons have been selected (from other)

The "pull_..." methods are similar to "select_..." but work the opposite way: they select shapes from the argument region rather than self. In a deep (hierarchical) context the output region will be hierarchically aligned with self, so the "pull_..." methods provide a way for re-hierarchization.

Merged semantics applies for this method (see merged_semantics= of merged semantics)

This method has been introduced in version 0.26.1

(2) Signature: *[const]* Edges **pull_interacting** (const Edges other)

Description: Returns all edges of "other" which are interacting with polygons of this edge set

Returns: The edge collection after the edges have been selected (from other)

See the other pull_interacting version for more details.

Merged semantics applies for this method (see merged_semantics= of merged semantics)

This method has been introduced in version 0.26.1

remove_properties

Signature: void **remove_properties**

Description: Removes properties for the given container.

This will remove all properties on the given container.

This method has been introduced in version 0.28.4.

select_inside

(1) Signature: Edges **select_inside** (const Edges other)

Description: Selects the edges from this edge collection which are inside (completely covered by) edges from the other edge collection

Returns: The edge collection after the edges have been selected (self)

This method has been introduced in version 0.28.

(2) Signature: Edges **select_inside** (const Region other)

Description: Selects the edges from this edge collection which are inside (completely covered by) polygons from the region

Returns: The edge collection after the edges have been selected (self)

This method has been introduced in version 0.28.

select_inside_part

Signature: Edges **select_inside_part** (const Region other)

Description: Selects the parts of the edges from this edge collection which are inside the polygons of the given region

Returns: The edge collection after the edges have been selected (self)

This operation selects the parts of the edges which are inside the given region. This functionality is similar to the '&=' operator, but edges on the borders of the polygons are not included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

This method has been introduced in version 0.24.

select_interacting

(1) Signature: Edges **select_interacting** (const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the edges from this edge collection which overlap or touch edges from the other edge collection

Returns: The edge collection after the edges have been selected (self)

This is the in-place version of interacting - i.e. self is modified rather than a new collection is returned.

'min_count' and 'max_count' have been introduced in version 0.29.

(2) Signature: Edges **select_interacting** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the edges from this edge collection which overlap or touch polygons from the region

Returns: The edge collection after the edges have been selected (self)

This is the in-place version of interacting - i.e. self is modified rather than a new collection is returned.

'min_count' and 'max_count' have been introduced in version 0.29.

select_not_inside

(1) Signature: Edges **select_not_inside** (const Edges other)

Description: Selects the edges from this edge collection which are not inside (completely covered by) edges from the other edge collection

Returns: The edge collection after the edges have been selected (self)

This method has been introduced in version 0.28.

(2) Signature: Edges **select_not_inside** (const Region other)

Description: Selects the edges from this edge collection which are not inside (completely covered by) polygons from the region

Returns: The edge collection after the edges have been selected (self)

This method has been introduced in version 0.28.

select_not_interacting

(1) Signature: Edges **select_not_interacting** (const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the edges from this edge collection which do not overlap or touch edges from the other edge collection

Returns: The edge collection after the edges have been selected (self)

This is the in-place version of not_interacting - i.e. self is modified rather than a new collection is returned.

'min_count' and 'max_count' have been introduced in version 0.29.

(2) Signature: Edges **select_not_interacting** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the edges from this edge collection which do not overlap or touch polygons from the region

Returns: The edge collection after the edges have been selected (self)

This is the in-place version of not_interacting - i.e. self is modified rather than a new collection is returned.

'min_count' and 'max_count' have been introduced in version 0.29.

select_not_outside

(1) Signature: Edges **select_not_outside** (const Edges other)

Description: Selects the edges from this edge collection which are not outside (completely covered by) edges from the other edge collection

Returns: The edge collection after the edges have been selected (self)

This method has been introduced in version 0.28.

(2) Signature: Edges **select_not_outside** (const Region other)

Description: Selects the edges from this edge collection which are not outside (completely covered by) polygons from the region

Returns: The edge collection after the edges have been selected (self)

This method has been introduced in version 0.28.

select_outside

(1) Signature: Edges **select_outside** (const Edges other)

Description: Selects the edges from this edge collection which are outside (completely covered by) edges from the other edge collection

Returns: The edge collection after the edges have been selected (self)

This method has been introduced in version 0.28.

(2) Signature: Edges **select_outside** (const Region other)

Description: Selects the edges from this edge collection which are outside (completely covered by) polygons from the region

Returns: The edge collection after the edges have been selected (self)

This method has been introduced in version 0.28.

select_outside_part

Signature: Edges **select_outside_part** (const Region other)

Description: Selects the parts of the edges from this edge collection which are outside the polygons of the given region

Returns: The edge collection after the edges have been selected (self)

This operation selects the parts of the edges which are not inside the given region. This functionality is similar to the '-=' operator, but edges on the borders of the polygons are included in the edge set. As a side effect, the edges are made non-intersecting by introducing cut points where edges intersect.

This method has been introduced in version 0.24.

separation_check

Signature: *[const]* EdgePairs **separation_check** (const Edges other, int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs an overlap check with options

d:	The minimum distance for which the edges are checked
other:	The other edge collection against which to check
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The threshold angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper threshold of the projected length of one edge onto another
zero_distance_mode:	Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants. Use nil for this value to select the default (Euclidian metrics).

"ignore_angle" specifies the angle threshold of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one threshold, pass nil to the respective value.

'zero_distance_mode' has been added in version 0.29.

size

Signature: *[const]* unsigned long **size**

Description: Returns the (flat) number of edges in the edge collection

Use of this method is deprecated. Use count instead

This returns the number of raw edges (not merged edges if merged semantics is enabled). The count is computed 'as if flat', i.e. edges inside a cell are multiplied by the number of times a cell is instantiated.

Starting with version 0.27, the method is called 'count' for consistency with Region. 'size' is still provided as an alias.

Python specific notes:
This method is also available as 'len(object)'.

space_check

Signature: *[const]* EdgePairs **space_check** (int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs a space check with options

d:	The minimum distance for which the edges are checked
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The threshold angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper threshold of the projected length of one edge onto another
zero_distance_mode:	Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the space check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants. Use nil for this value to select the default (Euclidian metrics).

"ignore_angle" specifies the angle threshold of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one threshold, pass nil to the respective value.

'zero_distance_mode' has been added in version 0.29.

split_inside

(1) Signature: *[const]* Edges[] **split_inside** (const Edges other)

Description: Selects the edges from this edge collection which are and are not inside (completely covered by) edges from the other collection

Returns:	A two-element list of edge collections (first: inside, second: non-inside)

This method provides a faster way to compute both inside and non-inside edges compared to using separate methods. It has been introduced in version 0.28.

(2) Signature: *[const]* Edges[] **split_inside** (const Region other)

Description: Selects the edges from this edge collection which are and are not inside (completely covered by) polygons from the other region

Returns:	A two-element list of edge collections (first: inside, second: non-inside)

This method provides a faster way to compute both inside and non-inside edges compared to using separate methods. It has been introduced in version 0.28.

split_interacting

(1) Signature: *[const]* Edges[] **split_interacting** (const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the edges from this edge collection which do and do not interact with edges from the other collection

Returns: A two-element list of edge collections (first: interacting, second: non-interacting)

This method provides a faster way to compute both interacting and non-interacting edges compared to using separate methods. It has been introduced in version 0.28. 'min_count' and 'max_count' have been introduced in version 0.29.

(2) Signature: *[const]* Edges[] **split_interacting** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the edges from this edge collection which do and do not interact with polygons from the other region

Returns: A two-element list of edge collections (first: interacting, second: non-interacting)

This method provides a faster way to compute both interacting and non-interacting edges compared to using separate methods. It has been introduced in version 0.28. 'min_count' and 'max_count' have been introduced in version 0.29.

split_outside

(1) Signature: *[const]* Edges[] **split_outside** (const Edges other)

Description: Selects the edges from this edge collection which are and are not outside (completely covered by) edges from the other collection

Returns: A two-element list of edge collections (first: outside, second: non-outside)

This method provides a faster way to compute both outside and non-outside edges compared to using separate methods. It has been introduced in version 0.28.

(2) Signature: *[const]* Edges[] **split_outside** (const Region other)

Description: Selects the edges from this edge collection which are and are not outside (completely covered by) polygons from the other region

Returns: A two-element list of edge collections (first: outside, second: non-outside)

This method provides a faster way to compute both outside and non-outside edges compared to using separate methods. It has been introduced in version 0.28.

start_segments

Signature: *[const]* Edges **start_segments** (unsigned int length, double fraction)

Description: Returns edges representing a part of the edge after the start point

Returns: A new collection of edges representing the start part

This method allows one to specify the length of these segments in a twofold way: either as a fixed length or by specifying a fraction of the original length:

```
edges = ...  # An edge collection
edges.start_segments(100, 0.0)    # All segments have a length of 100 DBU
```

```
edges.start_segments(0, 50.0)     # All segments have a length of half the
 original length
edges.start_segments(100, 50.0)   # All segments have a length of half the
 original length
                                  # or 100 DBU, whichever is larger
```

It is possible to specify 0 for both values. In this case, degenerated edges (points) are delivered which specify the start positions of the edges but can't participate in some functions.

swap

Signature: void **swap** (Edges other)

Description: Swap the contents of this edge collection with the contents of another one

This method is useful to avoid excessive memory allocation in some cases. For managed memory languages such as Ruby, those cases will be rare.

to_s

(1) Signature: *[const]* string **to_s**

Description: Converts the edge collection to a string

The length of the output is limited to 20 edges to avoid giant strings on large regions. For full output use "to_s" with a maximum count parameter.

Python specific notes:
This method is also available as 'str(object)'.

(2) Signature: *[const]* string **to_s** (unsigned long max_count)

Description: Converts the edge collection to a string

This version allows specification of the maximum number of edges contained in the string.

transform

(1) Signature: Edges **transform** (const Trans t)

Description: Transform the edge collection (modifies self)

t:	The transformation to apply.
Returns:	The transformed edge collection.

Transforms the edge collection with the given transformation. This version modifies the edge collection and returns a reference to self.

(2) Signature: Edges **transform** (const ICplxTrans t)

Description: Transform the edge collection with a complex transformation (modifies self)

t:	The transformation to apply.
Returns:	The transformed edge collection.

Transforms the edge collection with the given transformation. This version modifies the edge collection and returns a reference to self.

(3) Signature: Edges **transform** (const IMatrix2d t)

Description: Transform the edge collection (modifies self)

t:	The transformation to apply.
Returns:	The transformed edge collection.

Transforms the edge collection with the given 2d matrix transformation. This version modifies the edge collection and returns a reference to self.

This variant has been introduced in version 0.27.

(4) Signature: Edges **transform** (const IMatrix3d t)

Description: Transform the edge collection (modifies self)

t: The transformation to apply.

Returns: The transformed edge collection.

Transforms the edge collection with the given 3d matrix transformation. This version modifies the edge collection and returns a reference to self.

This variant has been introduced in version 0.27.

transform_icplx

Signature: Edges **transform_icplx** (const ICplxTrans t)

Description: Transform the edge collection with a complex transformation (modifies self)

t: The transformation to apply.

Returns: The transformed edge collection.

Use of this method is deprecated. Use transform instead

Transforms the edge collection with the given transformation. This version modifies the edge collection and returns a reference to self.

transformed

(1) Signature: *[const]* Edges **transformed** (const Trans t)

Description: Transform the edge collection

t: The transformation to apply.

Returns: The transformed edge collection.

Transforms the edge collection with the given transformation. Does not modify the edge collection but returns the transformed edge collection.

(2) Signature: *[const]* Edges **transformed** (const ICplxTrans t)

Description: Transform the edge collection with a complex transformation

t: The transformation to apply.

Returns: The transformed edge collection.

Transforms the edge collection with the given complex transformation. Does not modify the edge collection but returns the transformed edge collection.

(3) Signature: *[const]* Edges **transformed** (const IMatrix2d t)

Description: Transform the edge collection

t: The transformation to apply.

Returns: The transformed edge collection.

Transforms the edge collection with the given 2d matrix transformation. Does not modify the edge collection but returns the transformed edge collection.

This variant has been introduced in version 0.27.

(4) Signature: *[const]* Edges **transformed** (const IMatrix3d t)

Description: Transform the edge collection

t: The transformation to apply.

Returns: The transformed edge collection.

Transforms the edge collection with the given 3d matrix transformation. Does not modify the edge collection but returns the transformed edge collection.

This variant has been introduced in version 0.27.

transformed_icplx

Signature: *[const]* Edges **transformed_icplx** (const ICplxTrans t)

Description: Transform the edge collection with a complex transformation

t:	The transformation to apply.
Returns:	The transformed edge collection.

Use of this method is deprecated. Use transformed instead

Transforms the edge collection with the given complex transformation. Does not modify the edge collection but returns the transformed edge collection.

width_check

Signature: *[const]* EdgePairs **width_check** (int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs a width check with options

d:	The minimum width for which the edges are checked
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The threshold angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper threshold of the projected length of one edge onto another
zero_distance_mode:	Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants. Use nil for this value to select the default (Euclidian metrics).

"ignore_angle" specifies the angle threshold of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one threshold, pass nil to the respective value.

'zero_distance_mode' has been added in version 0.29.

with_angle

(1) Signature: *[const]* Edges **with_angle** (double angle, bool inverse)

Description: Filters the edges by orientation

Filters the edges in the edge collection by orientation. If "inverse" is false, only edges which have the given angle to the x-axis are returned. If "inverse" is true, edges not having the given angle are returned.

This will select horizontal edges:

```
horizontal = edges.with_angle(0, false)
```

(2) Signature: *[const]* Edges **with_angle** (double min_angle, double max_angle, bool inverse, bool include_min_angle = true, bool include_max_angle = false)

Description: Filters the edges by orientation

Filters the edges in the edge collection by orientation. If "inverse" is false, only edges which have an angle to the x-axis larger or equal to "min_angle" (depending on "include_min_angle") and equal or less than "max_angle" (depending on "include_max_angle") are returned. If "inverse" is true, edges which do not conform to this criterion are returned.

With "include_min_angle" set to true (the default), the minimum angle is included in the criterion while with false, the minimum angle itself is not included. Same for "include_max_angle" where the default is false, meaning the maximum angle is not included in the range.

The two "include.." arguments have been added in version 0.27.

(3) Signature: *[const]* Edges **with_angle** (Edges::EdgeType type, bool inverse)

Description: Filters the edges by orientation type

Filters the edges in the edge collection by orientation. If "inverse" is false, only edges which have an angle of the given type are returned. If "inverse" is true, edges which do not conform to this criterion are returned.

This version allows specifying an edge type instead of an angle. Edge types include multiple distinct orientations and are specified using one of the OrthoEdges, DiagonalEdges or OrthoDiagonalEdges types.

This method has been added in version 0.28.

with_length

(1) Signature: *[const]* Edges **with_length** (unsigned int length, bool inverse)

Description: Filters the edges by length

Filters the edges in the edge collection by length. If "inverse" is false, only edges which have the given length are returned. If "inverse" is true, edges not having the given length are returned.

(2) Signature: *[const]* Edges **with_length** (variant min_length, variant max_length, bool inverse)

Description: Filters the edges by length

Filters the edges in the edge collection by length. If "inverse" is false, only edges which have a length larger or equal to "min_length" and less than "max_length" are returned. If "inverse" is true, edges not having a length less than "min_length" or larger or equal than "max_length" are returned.

If you don't want to specify a lower or upper limit, pass nil to that parameter.

write

Signature: *[const]* void **write** (string filename)

Description: Writes the region to a file

This method is provided for debugging purposes. It writes the object to a flat layer 0/0 in a single top cell.

This method has been introduced in version 0.29.

xor

Signature: *[const]* Edges **xor** (const Edges other)

Description: Returns the boolean XOR between self and the other edge collection

Returns: The result of the boolean XOR operation

The boolean XOR operation will return all parts of the edges in this and the other collection except the parts where both are coincident. The result will be a merged edge collection.

The 'xor' alias has been introduced in version 0.28.12.

xor_with

Signature: Edges **xor_with** (const Edges other)

Description: Performs the boolean XOR between self and the other edge collection in-place (modifying self)

Returns: The edge collection after modification (self)

The boolean XOR operation will return all parts of the edges in this and the other collection except the parts where both are coincident. The result will be a merged edge collection.

Note that in Ruby, the '^=' operator actually does not exist, but is emulated by '^' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'xor_with' instead.

The 'xor_with' alias has been introduced in version 0.28.12.

|

Signature: *[const]* Edges **|** (const Edges other)

Description: Returns the boolean OR between self and the other edge set

Returns: The resulting edge collection

The boolean OR is implemented by merging the edges of both edge sets. To simply join the edge collections without merging, the + operator is more efficient. The 'or' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'or_' in Python.

|=

Signature: Edges **|=** (const Edges other)

Description: Performs the boolean OR between self and the other edge set in-place (modifying self)

Returns: The edge collection after modification (self)

The boolean OR is implemented by merging the edges of both edge sets. To simply join the edge collections without merging, the + operator is more efficient. Note that in Ruby, the '|=' operator actually does not exist, but is emulated by '|' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'or_with' instead.

The 'or_with' alias has been introduced in version 0.28.12.

4.48. API reference - Class Edges::EdgeType

Notation used in Ruby API documentation

Module: db

Description: This enum specifies the edge type for edge angle filters.

This class is equivalent to the class Edges::EdgeType

This enum was introduced in version 0.28.

Public constructors

new Edges::EdgeType ptr	**new**	(int i)	Creates an enum from an integer value
new Edges::EdgeType ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const Edges::EdgeType other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const Edges::EdgeType other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const Edges::EdgeType other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	Edges::EdgeType	**DiagonalEdges**	Diagonal edges are selected (-45 and 45 degree)
[static,const]	Edges::EdgeType	**OrthoDiagonalEdges**	Diagonal or orthogonal edges are selected (0, 90, -45 and 45 degree)
[static,const]	Edges::EdgeType	**OrthoEdges**	Horizontal and vertical edges are selected

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const Edges::EdgeType other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const Edges::EdgeType other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const Edges::EdgeType other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
DiagonalEdges	**Signature**: *[static,const]* Edges::EdgeType **DiagonalEdges** **Description**: Diagonal edges are selected (-45 and 45 degree) **Python specific notes:** The object exposes a readable attribute 'DiagonalEdges'. This is the getter.
OrthoDiagonalEdges	**Signature**: *[static,const]* Edges::EdgeType **OrthoDiagonalEdges** **Description**: Diagonal or orthogonal edges are selected (0, 90, -45 and 45 degree) **Python specific notes:** The object exposes a readable attribute 'OrthoDiagonalEdges'. This is the getter.
OrthoEdges	**Signature**: *[static,const]* Edges::EdgeType **OrthoEdges** **Description**: Horizontal and vertical edges are selected **Python specific notes:** The object exposes a readable attribute 'OrthoEdges'. This is the getter.
hash	**Signature**: *[const]* int **hash** **Description**: Gets the hash value from the enum **Python specific notes:** This method is also available as 'hash(object)'.
inspect	**Signature**: *[const]* string **inspect** **Description**: Converts an enum to a visual string **Python specific notes:** This method is also available as 'repr(object)'.

new

(1) Signature: *[static]* new Edges::EdgeType ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Edges::EdgeType ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.49. API reference - Class InstElement

Notation used in Ruby API documentation

Module: db

Description: An element in an instantiation path

This objects are used to reference a single instance in a instantiation path. The object is composed of a CellInstArray object (accessible through the cell_inst accessor) that describes the basic instance, which may be an array. The particular instance within the array can be further retrieved using the array_member_trans, specific_trans or specific_cplx_trans methods.

Public constructors

new InstElement ptr	**new**		Default constructor
new InstElement ptr	**new**	(const Instance inst)	Create an instance element from a single instance alone
new InstElement ptr	**new**	(const Instance inst, unsigned long a_index, unsigned long b_index)	Create an instance element from an array instance pointing into a certain array member

Public methods

[const]	bool	**!=**	(const InstElement b)	Inequality of two InstElement objects
[const]	bool	**<**	(const InstElement b)	Provides an order criterion for two InstElement objects
[const]	bool	**==**	(const InstElement b)	Equality of two InstElement objects
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	Trans	**array_member_trans**		Returns the transformation for this array member
	void	**assign**	(const InstElement other)	Assigns another object to self

[const]	const CellInstArray ptr	**cell_inst**	Accessor to the cell instance (array).
[const]	new InstElement ptr	**dup**	Creates a copy of self
[const]	long	**ia**	Returns the 'a' axis index for array instances
[const]	long	**ib**	Returns the 'b' axis index for array instances
[const]	Instance	**inst**	Gets the Instance object held in this instance path element.
[const]	unsigned long	**prop_id**	Accessor to the property attached to this instance.
[const]	ICplxTrans	**specific_cplx_trans**	Returns the specific complex transformation for this instance
[const]	Trans	**specific_trans**	Returns the specific transformation for this instance

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[static]	new InstElement ptr	**new_i**	(const Instance inst)	Use of this method is deprecated. Use new instead
[static]	new InstElement ptr	**new_iab**	(const Instance inst, unsigned long a_index, unsigned long b_index)	Use of this method is deprecated. Use new instead

Detailed description

!=	**Signature**: *[const]* bool **!=** (const InstElement b) **Description**: Inequality of two InstElement objects See the comments on the == operator.
<	**Signature**: *[const]* bool **<** (const InstElement b) **Description**: Provides an order criterion for two InstElement objects Note: this operator is just provided to establish any order, not a particular one.

==	**Signature**: *[const]* bool **==** (const InstElement b) **Description**: Equality of two InstElement objects Note: this operator returns true if both instance elements refer to the same instance, not just identical ones.
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
array_member_trans	**Signature**: *[const]* Trans **array_member_trans** **Description**: Returns the transformation for this array member

The array member transformation is the one applicable in addition to the global transformation for the member selected from an array. If this instance is not an array instance, the specific transformation is a unit transformation without displacement.

assign

Signature: void **assign** (const InstElement other)

Description: Assigns another object to self

cell_inst

Signature: *[const]* const CellInstArray ptr **cell_inst**

Description: Accessor to the cell instance (array).

This method is equivalent to "self.inst.cell_inst" and provided for convenience.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new InstElement ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

ia

Signature: *[const]* long **ia**

Description: Returns the 'a' axis index for array instances

For instance elements describing one member of an array, this attribute will deliver the a axis index addressed by this element. See ib and array_member_trans for further attributes applicable to array members. If the element is a plain instance and not an array member, this attribute is a negative value.

This method has been introduced in version 0.25.

ib

Signature: *[const]* long **ib**

Description: Returns the 'b' axis index for array instances

For instance elements describing one member of an array, this attribute will deliver the a axis index addressed by this element. See ia and array_member_trans for further attributes applicable to array members. If the element is a plain instance and not an array member, this attribute is a negative value.

This method has been introduced in version 0.25.

inst

Signature: *[const]* Instance **inst**

Description: Gets the Instance object held in this instance path element.

This method has been added in version 0.24.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

(1) Signature: *[static]* new InstElement ptr **new**

Description: Default constructor

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new InstElement ptr **new** (const Instance inst)

Description: Create an instance element from a single instance alone

Starting with version 0.15, this method takes an Instance object (an instance reference) as the argument.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new InstElement ptr **new** (const Instance inst, unsigned long a_index, unsigned long b_index)

Description: Create an instance element from an array instance pointing into a certain array member

Starting with version 0.15, this method takes an Instance object (an instance reference) as the first argument.

Python specific notes:
This method is the default initializer of the object.

new_i

Signature: *[static]* new InstElement ptr **new_i** (const Instance inst)

Description: Create an instance element from a single instance alone

Use of this method is deprecated. Use new instead

Starting with version 0.15, this method takes an Instance object (an instance reference) as the argument.

Python specific notes:
This method is the default initializer of the object.

new_iab

Signature: *[static]* new InstElement ptr **new_iab** (const Instance inst, unsigned long a_index, unsigned long b_index)

Description: Create an instance element from an array instance pointing into a certain array member

Use of this method is deprecated. Use new instead

Starting with version 0.15, this method takes an Instance object (an instance reference) as the first argument.

Python specific notes:
This method is the default initializer of the object.

prop_id

Signature: *[const]* unsigned long **prop_id**

Description: Accessor to the property attached to this instance.

This method is equivalent to "self.inst.prop_id" and provided for convenience.

specific_cplx_trans

Signature: *[const]* ICplxTrans **specific_cplx_trans**

Description: Returns the specific complex transformation for this instance

The specific transformation is the one applicable for the member selected from an array. This is the effective transformation applied for this array member. array_member_trans gives the transformation applied additionally to the instances' global transformation (in other words, specific_cplx_trans = array_member_trans * cell_inst.cplx_trans).

specific_trans

Signature: *[const]* Trans **specific_trans**

Description: Returns the specific transformation for this instance

The specific transformation is the one applicable for the member selected from an array. This is the effective transformation applied for this array member. array_member_trans gives the transformation applied additionally to the instances' global transformation (in other words, specific_trans = array_member_trans * cell_inst.trans). This method delivers a simple transformation that does not include magnification components. To get these as well, use specific_cplx_trans.

4.50. API reference - Class LayerMapping

Notation used in Ruby API documentation

Module: db

Description: A layer mapping (source to target layout)

A layer mapping is an association of layers in two layouts forming pairs of layers, i.e. one layer corresponds to another layer in the other layout. The LayerMapping object describes the mapping of layers of a source layout A to a target layout B.

A layer mapping can be set up manually or using the methods create or create_full.

```
lm = RBA::LayerMapping::new
# explicit:
lm.map(2, 1)  # map layer index 2 of source to 1 of target
lm.map(7, 3)  # map layer index 7 of source to 3 of target
...
# or employing the specification identity:
lm.create(target_layout, source_layout)
# plus creating layers which don't exist in the target layout yet:
new_layers = lm.create_full(target_layout, source_layout)
```

A layer might not be mapped to another layer which basically means that there is no corresponding layer. Such layers will be ignored in operations using the layer mapping. Use create_full to ensure all layers of the source layout are mapped.

LayerMapping objects play a role mainly in the hierarchical copy or move operations of Layout. However, use is not restricted to these applications.

This class has been introduced in version 0.23.

Public constructors

new LayerMapping ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const LayerMapping other)	Assigns another object to self
	void	**clear**		Clears the mapping.

	void	**create**	(const Layout layout_a, const Layout layout_b)	Initialize the layer mapping from two layouts
	unsigned int[]	**create_full**	(Layout layout_a, const Layout layout_b)	Initialize the layer mapping from two layouts
[const]	new LayerMapping ptr	**dup**		Creates a copy of self
[const]	bool	**has_mapping?**	(unsigned int layer_index_b)	Determine if a layer in layout_b has a mapping to a layout_a layer.
[const]	unsigned int	**layer_mapping**	(unsigned int layer_index_b)	Determine layer mapping of a layout_b layer to the corresponding layout_a layer.
	void	**map**	(unsigned int layer_index_b, unsigned int layer_index_a)	Explicitly specify a mapping.
[const]	map<unsigned int,unsigned int>	**table**		Returns the mapping table.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const LayerMapping other)

Description: Assigns another object to self

clear

Signature: void **clear**

Description: Clears the mapping.

create

Signature: void **create** (const Layout layout_a, const Layout layout_b)

Description: Initialize the layer mapping from two layouts

layout_a:	The target layout
layout_b:	The source layout

The layer mapping is created by looking up each layer of layout_b in layout_a. All layers with matching specifications (LayerInfo) are mapped. Layouts without a layer/datatype/name specification will not be mapped. create_full is a version of this method which creates new layers in layout_a if no corresponding layer is found.

create_full

Signature: unsigned int[] **create_full** (Layout layout_a, const Layout layout_b)

Description: Initialize the layer mapping from two layouts

layout_a:	The target layout
layout_b:	The source layout
Returns:	A list of layers created

The layer mapping is created by looking up each layer of layout_b in layout_a. All layers with matching specifications (LayerInfo) are mapped. Layouts without a layer/datatype/name specification will not be mapped. Layers with a valid specification which are not found in layout_a are created there.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new LayerMapping ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

has_mapping?

Signature: *[const]* bool **has_mapping?** (unsigned int layer_index_b)

Description: Determine if a layer in layout_b has a mapping to a layout_a layer.

layer_index_b:	The index of the layer in layout_b whose mapping is requested.
Returns:	true, if the layer has a mapping

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

layer_mapping

Signature: *[const]* unsigned int **layer_mapping** (unsigned int layer_index_b)

Description: Determine layer mapping of a layout_b layer to the corresponding layout_a layer.

layer_index_b:	The index of the layer in layout_b whose mapping is requested.
Returns:	The corresponding layer in layout_a.

map

Signature: void **map** (unsigned int layer_index_b, unsigned int layer_index_a)

Description: Explicitly specify a mapping.

layer_index_b:	The index of the layer in layout B (the "source")
layer_index_a:	The index of the layer in layout A (the "target")

	Beside using the mapping generator algorithms provided through create and create_full, it is possible to explicitly specify layer mappings using this method.
new	**Signature**: *[static]* new LayerMapping ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
table	**Signature**: *[const]* map<unsigned int,unsigned int> **table** **Description**: Returns the mapping table. The mapping table is a dictionary where the keys are source layout layer indexes and the values are the target layout layer indexes. This method has been introduced in version 0.25.

4.51. API reference - Class LayerInfo

Notation used in Ruby API documentation

Module: db

Description: A structure encapsulating the layer properties

The layer properties describe how a layer is stored in a GDS2 or OASIS file for example. The LayerInfo object represents the storage properties that are attached to a layer in the database.

In general, a layer has either a layer and a datatype number (in GDS2), a name (for example in DXF or CIF) or both (in OASIS). In the latter case, the primary identification is through layer and datatype number and the name is some annotation attached to it. A LayerInfo object which specifies just a name returns true on is_named?. The LayerInfo object can also specify an anonymous layer (use LayerInfo#new without arguments). Such a layer will not be stored when saving the layout. They can be employed for temporary layers for example. Use LayerInfo#anonymous? to test whether a layer does not have a specification.

The LayerInfo is used for example in Layout#insert_layer to specify the properties of the new layer that will be created. The is_equivalent? method compares two LayerInfo objects using the layer and datatype numbers with a higher priority over the name.

Public constructors

new LayerInfo ptr	**new**		The default constructor.
new LayerInfo ptr	**new**	(int layer, int datatype)	The constructor for a layer/datatype pair.
new LayerInfo ptr	**new**	(string name)	The constructor for a named layer.
new LayerInfo ptr	**new**	(int layer, int datatype, string name)	The constructor for a named layer with layer and datatype.

Public methods

[const]	bool	**!=**	(const LayerInfo b)	Compares two layer info objects
[const]	bool	**==**	(const LayerInfo b)	Compares two layer info objects
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	bool	**anonymous?**		Returns true, if the layer has no specification (i.e. is created by the default constructor).

	void	**assign**	(const LayerInfo other)	Assigns another object to self
[const]	int	**datatype**		Gets the datatype
	void	**datatype=**	(int datatype)	Set the datatype
[const]	new LayerInfo ptr	**dup**		Creates a copy of self
[const]	unsigned long	**hash**		Computes a hash value
[const]	bool	**is_equivalent?**	(const LayerInfo b)	Equivalence of two layer info objects
[const]	bool	**is_named?**		Returns true, if the layer is purely specified by name.
[const]	int	**layer**		Gets the layer number
	void	**layer=**	(int layer)	Sets the layer number
[const]	string	**name**		Gets the layer name
	void	**name=**	(string name)	Set the layer name
[const]	string	**to_s**	(bool as_target = false)	Convert the layer info object to a string

Public static methods and constants

LayerInfo	**from_string**	(string s, bool as_target = false)	Create a layer info object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const LayerInfo b)

Description: Compares two layer info objects

Returns: True, if both are not equal

This method was added in version 0.18.

==

Signature: *[const]* bool **==** (const LayerInfo b)

Description: Compares two layer info objects

Returns: True, if both are equal

This method was added in version 0.18.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

anonymous?

Signature: *[const]* bool **anonymous?**

Description: Returns true, if the layer has no specification (i.e. is created by the default constructor).

Returns: True, if the layer does not have any specification.

This method was added in version 0.23.

assign

Signature: void **assign** (const LayerInfo other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

datatype

Signature: *[const]* int **datatype**

Description: Gets the datatype

Python specific notes:
The object exposes a readable attribute 'datatype'. This is the getter.

datatype=

Signature: void **datatype=** (int datatype)

Description: Set the datatype

Python specific notes:
The object exposes a writable attribute 'datatype'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new LayerInfo ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_string

Signature: *[static]* LayerInfo **from_string** (string s, bool as_target = false)

Description: Create a layer info object from a string

The: string

Returns: The LayerInfo object

If 'as_target' is true, relative specifications such as '*+1' for layer or datatype are permitted.

This method will take strings as produced by to_s and create a LayerInfo object from them. The format is either "layer", "layer/datatype", "name" or "name (layer/datatype)".

This method was added in version 0.23. The 'as_target' argument has been added in version 0.26.5.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given layer info object. This method enables layer info objects as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_equivalent?

Signature: *[const]* bool **is_equivalent?** (const LayerInfo b)

Description: Equivalence of two layer info objects

Returns: True, if both are equivalent

First, layer and datatype are compared. The name is of second order and used only if no layer or datatype is given for one of the operands. This is basically a weak comparison that reflects the search preferences. It is the basis for Layout#find_layer. Here are some examples:

```
# no match as layer/datatypes or names differ:
RBA::LayerInfo::new(1, 17).is_equivalent?(RBA::LayerInfo::new(1, 18)) -> false
RBA::LayerInfo::new('metal1').is_equivalent?(RBA::LayerInfo::new('m1')) ->
 false
# exact match for numbered or named layers:
RBA::LayerInfo::new(1, 17).is_equivalent?(RBA::LayerInfo::new(1, 17)) -> true
RBA::LayerInfo::new('metal1').is_equivalent?(RBA::LayerInfo::new('metal1')) ->
 true
# match as names are second priority over layer/datatypes:
RBA::LayerInfo::new(1, 17, 'metal1').is_equivalent?(RBA::LayerInfo::new(1, 17,
 'm1')) -> true
# match as name matching is fallback:
RBA::LayerInfo::new(1, 17, 'metal1').is_equivalent?
(RBA::LayerInfo::new('metal1')) -> true
# no match as neither names or layer/datatypes match:
RBA::LayerInfo::new(1, 17, 'metal1').is_equivalent?(RBA::LayerInfo::new('m1'))
 -> false
```

This method was added in version 0.18 and modified to compare non-named vs. named layers in version 0.28.11.

is_named?

Signature: *[const]* bool **is_named?**

Description: Returns true, if the layer is purely specified by name.

Returns: True, if no layer or datatype is given.

This method was added in version 0.18.

layer

Signature: *[const]* int **layer**

Description: Gets the layer number

Python specific notes:
The object exposes a readable attribute 'layer'. This is the getter.

layer=

Signature: void **layer=** (int layer)

Description: Sets the layer number

Python specific notes:
The object exposes a writable attribute 'layer'. This is the setter.

name

Signature: *[const]* string **name**

Description: Gets the layer name

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string name)

Description: Set the layer name

The name is set on OASIS input for example, if the layer has a name.

Python specific notes:
The object exposes a writable attribute 'name'. This is the setter.

new

(1) Signature: *[static]* new LayerInfo ptr **new**

Description: The default constructor.

Creates a default LayerInfo object.

This method was added in version 0.18.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new LayerInfo ptr **new** (int layer, int datatype)

Description: The constructor for a layer/datatype pair.

layer: The layer number

datatype: The datatype number

Creates a LayerInfo object representing a layer and datatype.

This method was added in version 0.18.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new LayerInfo ptr **new** (string name)

Description: The constructor for a named layer.

name: The name

Creates a LayerInfo object representing a named layer.

This method was added in version 0.18.

Python specific notes:

This method is the default initializer of the object.

(4) Signature: *[static]* new LayerInfo ptr **new** (int layer, int datatype, string name)

Description: The constructor for a named layer with layer and datatype.

layer:	The layer number
datatype:	The datatype number
name:	The name

Creates a LayerInfo object representing a named layer with layer and datatype.

This method was added in version 0.18.

Python specific notes:
This method is the default initializer of the object.

to_s

Signature: *[const]* string **to_s** (bool as_target = false)

Description: Convert the layer info object to a string

Returns:	The string

If 'as_target' is true, wildcard and relative specifications are formatted such such.

This method was added in version 0.18. The 'as_target' argument has been added in version 0.26.5.

Python specific notes:
This method is also available as 'str(object)'.

4.52. API reference - Class Layout

Notation used in Ruby API documentation

Module: db

Description: The layout object

This object represents a layout. The layout object contains the cell hierarchy and adds functionality for managing cell names and layer names. The cell hierarchy can be changed by adding cells and cell instances. Cell instances will virtually put the content of a cell into another cell. Many cell instances can be put into a cell thus forming repetitions of the cell content. This process can be repeated over multiple levels. In effect a cell graphs is created with parent cells and child cells. The graph must not be recursive, so there is at least one top cell, which does not have a parent cell. Multiple top cells can be present.

Layout is the very basic class of the layout database. It has a rich set of methods to manipulate and query the layout hierarchy, the geometrical objects, the meta information and other features of the layout database. For a discussion of the basic API and the related classes see The Database API.

Usually layout objects have already been created by KLayout's application core. You can address such a layout via the CellView object inside the LayoutView class. For example:

```
active_layout = RBA::CellView::active.layout
puts "Top cell of current layout is #{active_layout.top_cell.name}"
```

However, a layout can also be used standalone:

```
layout = RBA::Layout::new
cell = layout.create_cell("TOP")
layer = layout.layer(RBA::LayerInfo::new(1, 0))
cell.shapes(layer).insert(RBA::Box::new(0, 0, 1000, 1000))
layout.write("single_rect.gds")
```

Public constructors

new Layout ptr	**new**	(Manager manager)	Creates a layout object attached to a manager
new Layout ptr	**new**		Creates a layout object
new Layout ptr	**new**	(bool editable, Manager manager)	Creates a layout object attached to a manager
new Layout ptr	**new**	(bool editable)	Creates a layout object

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference

	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
	unsigned int	add_lib_cell	(Library ptr library, unsigned int lib_cell_index)	Imports a cell from the library
	void	add_meta_info	(const LayoutMetaInfo info)	Adds meta information to the layout
	unsigned int	add_pcell_variant	(unsigned int pcell_id, map<string,variant> parameters)	Creates a PCell variant for the given PCell ID with the parameters given as a name/value dictionary
	unsigned int	add_pcell_variant	(unsigned int pcell_id, variant[] parameters)	Creates a PCell variant for the given PCell ID with the given parameters
	unsigned int	add_pcell_variant	(Library ptr library, unsigned int pcell_id, map<string,variant> parameters)	Creates a PCell variant for a PCell located in an external library with the parameters given as a name/value dictionary
	unsigned int	add_pcell_variant	(Library ptr library, unsigned int pcell_id, variant[] parameters)	Creates a PCell variant for a PCell located in an external library
	void	assign	(const Layout other)	Assigns another object to self
[const]	RecursiveShapeIterator	begin_shapes	(const Cell ptr cell, unsigned int layer)	Delivers a recursive shape iterator for the shapes below the given cell on the given layer
	Cell ptr	cell	(string name)	Gets a cell object from the cell name
	Cell ptr	cell	(unsigned int i)	Gets a cell object from the cell index
[const]	string	cell_name	(unsigned int index)	Gets the name for a cell with the given index
[const]	unsigned int	cells		Returns the number of cells
	Cell ptr[]	cells	(string name_filter)	Gets the cell objects for a given name filter
	void	cleanup	(unsigned int[] cell_indexes_to_keep = [])	Cleans up the layout
	void	clear		Clears the layout
	void	clear_all_meta_info		Clears all meta information of the layout (cell specific and global)
	void	clear_layer	(unsigned int layer_index)	Clears a layer
	void	clear_layer	(unsigned int layer_index, unsigned int flags)	Clears a layer (given shape types only)

	void	**clear_meta_info**		Clears the meta information of the layout
	unsigned int	**clip**	(unsigned int cell, const Box box)	Clips the given cell by the given rectangle and produce a new cell with the clip
	unsigned int	**clip**	(unsigned int cell, const DBox box)	Clips the given cell by the given rectangle and produce a new cell with the clip
	Cell ptr	**clip**	(const Cell cell, const Box box)	Clips the given cell by the given rectangle and produce a new cell with the clip
	Cell ptr	**clip**	(const Cell cell, const DBox box)	Clips the given cell by the given rectangle and produce a new cell with the clip
[const]	unsigned int	**clip_into**	(unsigned int cell, Layout ptr target, const Box box)	Clips the given cell by the given rectangle and produce a new cell with the clip
[const]	unsigned int	**clip_into**	(unsigned int cell, Layout ptr target, const DBox box)	Clips the given cell by the given rectangle and produce a new cell with the clip
[const]	Cell ptr	**clip_into**	(const Cell cell, Layout ptr target, const Box box)	Clips the given cell by the given rectangle and produce a new cell with the clip
[const]	Cell ptr	**clip_into**	(const Cell cell, Layout ptr target, const DBox box)	Clips the given cell by the given rectangle and produce a new cell with the clip
	unsigned int	**convert_cell_to_static**	(unsigned int cell_index)	Converts a PCell or library cell to a usual (static) cell
	void	**copy_layer**	(unsigned int src, unsigned int dest)	Copies a layer
	void	**copy_layer**	(unsigned int src, unsigned int dest, unsigned int flags)	Copies a layer (selected shape types only)
	void	**copy_meta_info**	(const Layout other)	Copies the meta information from the other layout into this layout
	void	**copy_meta_info**	(const Layout other, const CellMapping cm)	Copies the meta information from the other layout into this layout for the cells given by the cell mapping
	void	**copy_tree_shapes**	(const Layout source_layout, const CellMapping cell_mapping)	Copies the shapes for all given mappings in the CellMapping object
	void	**copy_tree_shapes**	(const Layout source_layout, const CellMapping cell_mapping, const LayerMapping layer_mapping)	Copies the shapes for all given mappings in the CellMapping object using the given layer mapping

	Cell ptr	**create_cell**	(string name)	Creates a cell with the given name
	Cell ptr	**create_cell**	(string pcell_name, map<string,variant> params)	Creates a cell as a PCell variant for the PCell with the given name
	Cell ptr	**create_cell**	(string name, string lib_name)	Creates a cell with the given name
	Cell ptr	**create_cell**	(string pcell_name, string lib_name, map<string,variant> params)	Creates a cell for a PCell with the given PCell name from the given library
[const]	double	**dbu**		Gets the database unit
	void	**dbu=**	(double dbu)	Sets the database unit
	void	**delete_cell**	(unsigned int cell_index)	Deletes a cell
	void	**delete_cell_rec**	(unsigned int cell_index)	Deletes a cell plus all subcells
	void	**delete_cells**	(unsigned int[] cell_index_list)	Deletes multiple cells
	void	**delete_layer**	(unsigned int layer_index)	Deletes a layer
	void	**delete_property**	(variant key)	Deletes the user property with the given key
[const]	new Layout ptr	**dup**		Creates a copy of self
[iter]	Cell	**each_cell**		Iterates the unsorted cell list
[iter]	unsigned int	**each_cell_bottom_up**		Iterates the bottom-up sorted cell list
[iter]	unsigned int	**each_cell_top_down**		begin iterator of the top-down sorted cell list
[const,iter]	LayoutMetaInfo	**each_meta_info**		Iterates over the meta information of the layout
[iter]	unsigned int	**each_top_cell**		Iterates the top cells
	void	**end_changes**		Cancels the "in changes" state (see "start_changes")
[const]	unsigned int	**error_layer**		Returns the index of the error layer
	variant	**find_layer**	(const LayerInfo info)	Finds a layer with the given properties
	variant	**find_layer**	(string name)	Finds a layer with the given name
	variant	**find_layer**	(int layer, int datatype)	Finds a layer with the given layer and datatype number
	variant	**find_layer**	(int layer, int datatype, string name)	Finds a layer with the given layer and datatype number and name
	void	**flatten**	(unsigned int cell_index,	Flattens the given cell

			int levels, bool prune)	
	void	**flatten_into**	(unsigned int source_cell_index, unsigned int target_cell_index, const ICplxTrans trans, int levels)	Flattens the given cell into another cell
[const]	LayerInfo	**get_info**	(unsigned int index)	Gets the info structure for a specified layer
[const]	unsigned int	**guiding_shape_layer**		Returns the index of the guiding shape layer
	bool	**has_cell?**	(string name)	Returns true if a cell with a given name exists
[const]	bool	**has_prop_id?**		Returns true, if the layout has user properties
	void	**insert**	(unsigned int cell_index, int layer, const Region region)	Inserts a region into the given cell and layer
	void	**insert**	(unsigned int cell_index, int layer, const Edges edges)	Inserts an edge collection into the given cell and layer
	void	**insert**	(unsigned int cell_index, int layer, const EdgePairs edge_pairs)	Inserts an edge pair collection into the given cell and layer
	void	**insert**	(unsigned int cell_index, int layer, const Texts texts)	Inserts an text collection into the given cell and layer
	unsigned int	**insert_layer**	(const LayerInfo props)	Inserts a new layer with the given properties
	void	**insert_layer_at**	(unsigned int index, const LayerInfo props)	Inserts a new layer with the given properties at the given index
	unsigned int	**insert_special_layer**	(const LayerInfo props)	Inserts a new special layer with the given properties
	void	**insert_special_layer_a**	(unsigned int index, const LayerInfo props)	Inserts a new special layer with the given properties at the given index
[const]	bool	**is_editable?**		Returns a value indicating whether the layout is editable.
[const]	bool	**is_free_layer?**	(unsigned int layer_index)	Returns true, if a layer index is a free (unused) layer index
[const]	bool	**is_special_layer?**	(unsigned int layer_index)	Returns true, if a layer index is a special layer index

[const]	bool	**is_valid_cell_index?**	(unsigned int cell_index)	Returns true, if a cell index is a valid index
[const]	bool	**is_valid_layer?**	(unsigned int layer_index)	Returns true, if a layer index is a valid normal layout layer index
	unsigned int	**layer**		Creates a new internal layer
	unsigned int	**layer**	(const LayerInfo info)	Finds or creates a layer with the given properties
	unsigned int	**layer**	(string name)	Finds or creates a layer with the given name
	unsigned int	**layer**	(int layer, int datatype)	Finds or creates a layer with the given layer and datatype number
	unsigned int	**layer**	(int layer, int datatype, string name)	Finds or creates a layer with the given layer and datatype number and name
[const]	unsigned int[]	**layer_indexes**		Gets a list of valid layer's indices
[const]	LayerInfo[]	**layer_infos**		Gets a list of valid layer's properties
[const]	unsigned int	**layers**		Returns the number of layers
[const]	Library ptr	**library**		Gets the library this layout lives in or nil if the layout is not part of a library
	void	**merge_meta_info**	(const Layout other)	Merges the meta information from the other layout into this layout
	void	**merge_meta_info**	(const Layout other, const CellMapping cm)	Merges the meta information from the other layout into this layout for the cells given by the cell mapping
	new LayoutMetaInfo ptr	**meta_info**	(string name)	Gets the meta information for a given name
	variant	**meta_info_value**	(string name)	Gets the meta information value for a given name
	void	**move_layer**	(unsigned int src, unsigned int dest)	Moves a layer
	void	**move_layer**	(unsigned int src, unsigned int dest, unsigned int flags)	Moves a layer (selected shape types only)
	void	**move_tree_shapes**	(Layout source_layout, const CellMapping cell_mapping)	Moves the shapes for all given mappings in the CellMapping object
	void	**move_tree_shapes**	(Layout source_layout, const CellMapping cell_mapping, const LayerMapping layer_mapping)	Moves the shapes for all given mappings in the CellMapping object using the given layer mapping

	unsigned int[]	**multi_clip**	(unsigned int cell, Box[] boxes)	Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.
	unsigned int[]	**multi_clip**	(unsigned int cell, DBox[] boxes)	Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.
	Cell ptr[]	**multi_clip**	(const Cell cell, Box[] boxes)	Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.
	Cell ptr[]	**multi_clip**	(const Cell cell, DBox[] boxes)	Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.
	unsigned int[]	**multi_clip_into**	(unsigned int cell, Layout ptr target, Box[] boxes)	Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.
	unsigned int[]	**multi_clip_into**	(unsigned int cell, Layout ptr target, DBox[] boxes)	Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.
	Cell ptr[]	**multi_clip_into**	(const Cell cell, Layout ptr target, Box[] boxes)	Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.
	Cell ptr[]	**multi_clip_into**	(const Cell cell, Layout ptr target, DBox[] boxes)	Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.
[const]	const PCellDeclaration ptr	**pcell_declaration**	(string name)	Gets a reference to the PCell declaration for the PCell with the given name
[const]	const PCellDeclaration ptr	**pcell_declaration**	(unsigned int pcell_id)	Gets a reference to the PCell declaration for the PCell with the given PCell ID.
[const]	unsigned int	**pcell_id**	(string name)	Gets the ID of the PCell with the given name
[const]	unsigned int[]	**pcell_ids**		Gets the IDs of the PCells registered in the layout
[const]	string[]	**pcell_names**		Gets the names of the PCells registered in the layout
[const]	unsigned long	**prop_id**		Gets the properties ID associated with the layout
	void	**prop_id=**	(unsigned long id)	Sets the properties ID associated with the layout
[const]	variant[]	**properties**	(unsigned long properties_id)	Gets the properties set for a given properties ID

	unsigned long	**properties_id**	(variant[] properties)	Gets the properties ID for a given properties set
	variant	**property**	(variant key)	Gets the user property with the given key
	void	**prune_cell**	(unsigned int cell_index, int levels)	Deletes a cell plus subcells not used otherwise
	void	**prune_subcells**	(unsigned int cell_index, int levels)	Deletes all sub cells of the cell which are not used otherwise down to the specified level of hierarchy
	LayerMap	**read**	(string filename)	Load the layout from the given file
	LayerMap	**read**	(string filename, const LoadLayoutOptions options)	Load the layout from the given file with options
	void	**refresh**		Calls Cell#refresh on all cells inside this layout
	unsigned int	**register_pcell**	(string name, PCellDeclaration ptr declaration)	Registers a PCell declaration under the given name
	void	**remove_meta_info**	(string name)	Removes meta information from the layout
	void	**rename_cell**	(unsigned int index, string name)	Renames the cell with given index
	void	**scale_and_snap**	(Cell cell, int grid, int mult, int div)	Scales and snaps the layout below a given cell by the given rational factor and snaps to the given grid
	void	**scale_and_snap**	(unsigned int cell_index, int grid, int mult, int div)	Scales and snaps the layout below a given cell by the given rational factor and snaps to the given grid
	void	**set_info**	(unsigned int index, const LayerInfo props)	Sets the info structure for a specified layer
	void	**set_property**	(variant key, variant value)	Sets the user property with the given key to the given value
	void	**start_changes**		Signals the start of an operation bringing the layout into invalid state
	void	**swap_layers**	(unsigned int a, unsigned int b)	Swap two layers
[const]	const Technology ptr	**technology**		Gets the Technology object of the technology this layout is associated with or nil if the layout is not associated with a technology
[const]	string	**technology_name**		Gets the name of the technology this layout is associated with

	void	technology_name=	(string name)	Sets the name of the technology this layout is associated with
	Cell ptr	top_cell		Returns the top cell object
	Cell ptr[]	top_cells		Returns the top cell objects
	void	transform	(const Trans trans)	Transforms the layout with the given transformation
	void	transform	(const ICplxTrans trans)	Transforms the layout with the given complex integer transformation
	void	transform	(const DTrans trans)	Transforms the layout with the given transformation, which is in micrometer units
	void	transform	(const DCplxTrans trans)	Transforms the layout with the given complex integer transformation, which is in micrometer units
[const]	bool	under_construction?		Returns true if the layout object is under construction
[const]	string	unique_cell_name	(string name)	Creates a new unique cell name from the given name
	void	update		Updates the internals of the layout
	void	write	(string filename, const SaveLayoutOptions options)	Writes the layout to a stream file
	void	write	(string filename)	Writes the layout to a stream file

Deprecated methods (protected, public, static, non-static and constructors)

	unsigned int	add_cell	(string name)	Use of this method is deprecated
[const]	RecursiveShapeIterator	begin_shapes	(unsigned int cell_index, unsigned int layer)	Use of this method is deprecated
[const]	RecursiveShapeIterator	begin_shapes_overlap	(unsigned int cell_index, unsigned int layer, Box region)	Use of this method is deprecated
[const]	RecursiveShapeIterator	begin_shapes_overlapping	(const Cell ptr cell_index, unsigned int layer, Box region)	Use of this method is deprecated
[const]	RecursiveShapeIterator	begin_shapes_overlap	(unsigned int cell_index, unsigned int layer, DBox region)	Use of this method is deprecated
[const]	RecursiveShapeIterator	begin_shapes_overlapping	(const Cell ptr cell, unsigned int layer, DBox region)	Use of this method is deprecated
[const]	RecursiveShapeIterator	begin_shapes_touchin	(unsigned int cell_index,	Use of this method is deprecated

			unsigned int layer, Box region)	
[const]	RecursiveShapeIterator	**begin_shapes_touching**	(const Cell ptr cell, unsigned int layer, Box region)	Use of this method is deprecated
[const]	RecursiveShapeIterator	**begin_shapes_touchir**	(unsigned int cell_index, unsigned int layer, DBox region)	Use of this method is deprecated
[const]	RecursiveShapeIterator	**begin_shapes_touching**	(const Cell ptr cell, unsigned int layer, DBox region)	Use of this method is deprecated
	unsigned int	**cell_by_name**	(string name)	Use of this method is deprecated
	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	unsigned int[]	**layer_indices**		Use of this method is deprecated. Use layer_indexes instead
	void	**write**	(string filename, bool gzip, const SaveLayoutOptions options)	Use of this method is deprecated

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_cell

Signature: unsigned int **add_cell** (string name)

Description: Adds a cell with the given name

Returns: The index of the newly created cell.

Use of this method is deprecated

From version 0.23 on this method is deprecated because another method exists which is more convenient because is returns a Cell object (create_cell).

add_lib_cell

Signature: unsigned int **add_lib_cell** (Library ptr library, unsigned int lib_cell_index)

Description: Imports a cell from the library

library: The reference to the library from which to import the cell

lib_cell_index: The index of the imported cell in the library

Returns: The cell index of the new proxy cell in this layout

This method imports the given cell from the library and creates a new proxy cell. The proxy cell acts as a pointer to the actual cell which still resides in the library (precisely: in library.layout). The name of the new cell will be the name of library cell.

This method has been introduced in version 0.22.

add_meta_info

Signature: void **add_meta_info** (const LayoutMetaInfo info)

Description: Adds meta information to the layout

See LayoutMetaInfo for details about layouts and meta information. This method has been introduced in version 0.25.

add_pcell_variant

(1) Signature: unsigned int **add_pcell_variant** (unsigned int pcell_id, map<string,variant> parameters)

Description: Creates a PCell variant for the given PCell ID with the parameters given as a name/value dictionary

Returns: The cell index of the pcell variant proxy cell

This method will create a PCell variant proxy for a local PCell definition. It will create the PCell variant for the given parameters. Note that this method does not allow one to create PCell instances for PCells located in a library. Use add_pcell_variant with the library parameter for that purpose. Unlike the variant using a list of parameters, this version allows specification of the parameters with a key/value dictionary. The keys are the parameter names as given by the PCell declaration.

The parameters are a sequence of variants which correspond to the parameters declared by the PCellDeclaration object.

The name of the new cell will be the name of the PCell. If a cell with that name already exists, a new unique name is generated.

This method has been introduced in version 0.22.

(2) Signature: unsigned int **add_pcell_variant** (unsigned int pcell_id, variant[] parameters)

Description: Creates a PCell variant for the given PCell ID with the given parameters

Returns: The cell index of the pcell variant proxy cell

This method will create a PCell variant proxy for a local PCell definition. It will create the PCell variant for the given parameters. Note that this method does not allow one to create PCell instances for PCells located in a library. Use add_pcell_variant with the library parameter for that purpose.

The parameters are a sequence of variants which correspond to the parameters declared by the PCellDeclaration object.

The name of the new cell will be the name of the PCell. If a cell with that name already exists, a new unique name is generated.

This method has been introduced in version 0.22.

(3) Signature: unsigned int **add_pcell_variant** (Library ptr library, unsigned int pcell_id, map<string,variant> parameters)

Description: Creates a PCell variant for a PCell located in an external library with the parameters given as a name/value dictionary

Returns: The cell index of the new proxy cell in this layout

This method will import a PCell from a library and create a variant for the given parameter set. Technically, this method creates a proxy to the library and creates the variant inside that library. Unlike the variant using a list of parameters, this version allows specification of the parameters with a key/value dictionary. The keys are the parameter names as given by the PCell declaration.

The parameters are a sequence of variants which correspond to the parameters declared by the PCellDeclaration object.

The name of the new cell will be the name of the PCell. If a cell with that name already exists, a new unique name is generated.

This method has been introduced in version 0.22.

(4) Signature: unsigned int **add_pcell_variant** (Library ptr library, unsigned int pcell_id, variant[] parameters)

Description: Creates a PCell variant for a PCell located in an external library

Returns: The cell index of the new proxy cell in this layout

This method will import a PCell from a library and create a variant for the given parameter set. Technically, this method creates a proxy to the library and creates the variant inside that library.

The parameters are a sequence of variants which correspond to the parameters declared by the PCellDeclaration object.

The name of the new cell will be the name of the PCell. If a cell with that name already exists, a new unique name is generated.

This method has been introduced in version 0.22.

assign

Signature: void **assign** (const Layout other)

Description: Assigns another object to self

begin_shapes

(1) Signature: *[const]* RecursiveShapeIterator **begin_shapes** (const Cell ptr cell, unsigned int layer)

Description: Delivers a recursive shape iterator for the shapes below the given cell on the given layer

cell:	The cell object of the initial (top) cell
layer:	The layer from which to get the shapes
Returns:	A suitable iterator

For details see the description of the RecursiveShapeIterator class. This version is convenience overload which takes a cell object instead of a cell index.

This method is deprecated. Use Cell#begin_shapes_rec instead.

This method has been added in version 0.24.

(2) Signature: *[const]* RecursiveShapeIterator **begin_shapes** (unsigned int cell_index, unsigned int layer)

Description: Delivers a recursive shape iterator for the shapes below the given cell on the given layer

cell_index:	The index of the initial (top) cell
layer:	The layer from which to get the shapes
Returns:	A suitable iterator

Use of this method is deprecated

For details see the description of the RecursiveShapeIterator class.

This method is deprecated. Use Cell#begin_shapes_rec instead.

This method has been added in version 0.18.

begin_shapes_overlapping

(1) Signature: *[const]* RecursiveShapeIterator **begin_shapes_overlapping** (unsigned int cell_index, unsigned int layer, Box region)

Description: Delivers a recursive shape iterator for the shapes below the given cell on the given layer using a region search

cell_index:	The index of the starting cell
layer:	The layer from which to get the shapes
region:	The search region
Returns:	A suitable iterator

Use of this method is deprecated

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box overlaps the given region.

This method is deprecated. Use Cell#begin_shapes_rec_overlapping instead.

This method has been added in version 0.18.

(2) Signature: *[const]* RecursiveShapeIterator **begin_shapes_overlapping** (const Cell ptr cell_index, unsigned int layer, Box region)

Description: Delivers a recursive shape iterator for the shapes below the given cell on the given layer using a region search

cell:	The cell object for the starting cell
layer:	The layer from which to get the shapes
region:	The search region
Returns:	A suitable iterator

Use of this method is deprecated

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box overlaps the given region. It is convenience overload which takes a cell object instead of a cell index.

This method is deprecated. Use Cell#begin_shapes_rec_overlapping instead.

This method has been added in version 0.24.

(3) Signature: *[const]* RecursiveShapeIterator **begin_shapes_overlapping** (unsigned int cell_index, unsigned int layer, DBox region)

Description: Delivers a recursive shape iterator for the shapes below the given cell on the given layer using a region search, the region given in micrometer units

cell_index:	The index of the starting cell
layer:	The layer from which to get the shapes
region:	The search region as a DBox object in micrometer units
Returns:	A suitable iterator

Use of this method is deprecated

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box overlaps the given region.

This method is deprecated. Use Cell#begin_shapes_rec_overlapping instead.

This variant has been added in version 0.25.

(4) Signature: *[const]* RecursiveShapeIterator **begin_shapes_overlapping** (const Cell ptr cell, unsigned int layer, DBox region)

Description: Delivers a recursive shape iterator for the shapes below the given cell on the given layer using a region search, the region given in micrometer units

cell:	The cell object for the starting cell
layer:	The layer from which to get the shapes
region:	The search region as a DBox object in micrometer units
Returns:	A suitable iterator

Use of this method is deprecated

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box overlaps the given region. It is convenience overload which takes a cell object instead of a cell index.

This method is deprecated. Use Cell#begin_shapes_rec_overlapping instead.

This variant has been added in version 0.25.

begin_shapes_touching

(1) Signature: *[const]* RecursiveShapeIterator **begin_shapes_touching** (unsigned int cell_index, unsigned int layer, Box region)

Description: Delivers a recursive shape iterator for the shapes below the given cell on the given layer using a region search

cell_index:	The index of the starting cell
layer:	The layer from which to get the shapes
region:	The search region
Returns:	A suitable iterator

Use of this method is deprecated

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box touches the given region.

This method is deprecated. Use Cell#begin_shapes_rec_touching instead.

This method has been added in version 0.18.

(2) Signature: *[const]* RecursiveShapeIterator **begin_shapes_touching** (const Cell ptr cell, unsigned int layer, Box region)

Description: Delivers a recursive shape iterator for the shapes below the given cell on the given layer using a region search

cell:	The cell object for the starting cell
layer:	The layer from which to get the shapes
region:	The search region
Returns:	A suitable iterator

Use of this method is deprecated

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box touches the given region. It is convenience overload which takes a cell object instead of a cell index.

This method is deprecated. Use Cell#begin_shapes_rec_touching instead.

This method has been added in version 0.24.

(3) Signature: *[const]* RecursiveShapeIterator **begin_shapes_touching** (unsigned int cell_index, unsigned int layer, DBox region)

Description: Delivers a recursive shape iterator for the shapes below the given cell on the given layer using a region search, the region given in micrometer units

cell_index:	The index of the starting cell
layer:	The layer from which to get the shapes
region:	The search region as a DBox object in micrometer units
Returns:	A suitable iterator

Use of this method is deprecated

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box touches the given region.

This method is deprecated. Use Cell#begin_shapes_rec_touching instead.

This variant has been added in version 0.25.

(4) Signature: *[const]* RecursiveShapeIterator **begin_shapes_touching** (const Cell ptr cell, unsigned int layer, DBox region)

Description: Delivers a recursive shape iterator for the shapes below the given cell on the given layer using a region search, the region given in micrometer units

cell:	The cell object for the starting cell
layer:	The layer from which to get the shapes
region:	The search region as a DBox object in micrometer units
Returns:	A suitable iterator

Use of this method is deprecated

For details see the description of the RecursiveShapeIterator class. This version gives an iterator delivering shapes whose bounding box touches the given region. It is convenience overload which takes a cell object instead of a cell index.

This method is deprecated. Use Cell#begin_shapes_rec_touching instead.

This variant has been added in version 0.25.

cell

(1) Signature: Cell ptr **cell** (string name)

Description: Gets a cell object from the cell name

name:	The cell name
Returns:	A reference to the cell (a Cell object)

If name is not a valid cell name, this method will return "nil". This method has been introduced in version 0.23 and replaces cell_by_name.

(2) Signature: Cell ptr **cell** (unsigned int i)

Description: Gets a cell object from the cell index

i:	The cell index
Returns:	A reference to the cell (a Cell object)

If the cell index is not a valid cell index, this method will raise an error. Use is_valid_cell_index? to test whether a given cell index is valid.

cell_by_name

Signature: unsigned int **cell_by_name** (string name)

Description: Gets the cell index for a given name

Use of this method is deprecated

Returns the cell index for the cell with the given name. If no cell with this name exists, an exception is thrown. From version 0.23 on, a version of the cell method is provided which returns a Cell object for the cell with the given name or "nil" if the name is not valid. This method replaces cell_by_name and has_cell?

cell_name

Signature: *[const]* string **cell_name** (unsigned int index)

Description: Gets the name for a cell with the given index

cells

(1) Signature: *[const]* unsigned int **cells**

Description: Returns the number of cells

Returns:	The number of cells (the maximum cell index)

(2) Signature: Cell ptr[] **cells** (string name_filter)

Description: Gets the cell objects for a given name filter

name_filter:	The cell name filter (glob pattern)
Returns:	A list of Cell object of the cells matching the pattern

This method has been introduced in version 0.27.3.

cleanup

Signature: void **cleanup** (unsigned int[] cell_indexes_to_keep = [])

Description: Cleans up the layout

This method will remove proxy objects that are no longer in use. After changing PCell parameters such proxy objects may still be present in the layout and are cached for later reuse. Usually they are cleaned up automatically, but in a scripting context it may be useful to clean up these cells explicitly.

Use 'cell_indexes_to_keep' for specifying a list of cell indexes of PCell variants or library proxies you don't want to be cleaned up.

This method has been introduced in version 0.25.

clear

Signature: void **clear**

Description: Clears the layout

Clears the layout completely.

clear_all_meta_info

Signature: void **clear_all_meta_info**

Description: Clears all meta information of the layout (cell specific and global)

See LayoutMetaInfo for details about layouts and meta information. This method has been introduced in version 0.28.16.

clear_layer

(1) Signature: void **clear_layer** (unsigned int layer_index)

Description: Clears a layer

layer_index:	The index of the layer to delete.

Clears the layer: removes all shapes.

This method was introduced in version 0.19.

(2) Signature: void **clear_layer** (unsigned int layer_index, unsigned int flags)

Description: Clears a layer (given shape types only)

layer_index:	The index of the layer to delete.
flags:	The type selector for the shapes to delete (see Shapes class, S... constants).

Clears the layer: removes all shapes for the given shape types.

This method was introduced in version 0.28.9.

clear_meta_info

Signature: void **clear_meta_info**

Description: Clears the meta information of the layout

See LayoutMetaInfo for details about layouts and meta information. This method has been introduced in version 0.28.8.

clip

(1) Signature: unsigned int **clip** (unsigned int cell, const Box box)

Description: Clips the given cell by the given rectangle and produce a new cell with the clip

cell: The cell index of the cell to clip

box: The clip box in database units

Returns: The index of the new cell

This method will cut a rectangular region given by the box from the given cell. The clip will be stored in a new cell whose index is returned. The clip will be performed hierarchically. The resulting cell will hold a hierarchy of child cells, which are potentially clipped versions of child cells of the original cell. This method has been added in version 0.21.

(2) Signature: unsigned int **clip** (unsigned int cell, const DBox box)

Description: Clips the given cell by the given rectangle and produce a new cell with the clip

cell: The cell index of the cell to clip

box: The clip box in micrometer units

Returns: The index of the new cell

This variant which takes a micrometer-unit box has been added in version 0.28.

(3) Signature: Cell ptr **clip** (const Cell cell, const Box box)

Description: Clips the given cell by the given rectangle and produce a new cell with the clip

cell: The cell reference of the cell to clip

box: The clip box in database units

Returns: The reference to the new cell

This variant which takes cell references instead of cell indexes has been added in version 0.28.

(4) Signature: Cell ptr **clip** (const Cell cell, const DBox box)

Description: Clips the given cell by the given rectangle and produce a new cell with the clip

cell: The cell reference of the cell to clip

box: The clip box in micrometer units

Returns: The reference to the new cell

This variant which takes a micrometer-unit box and cell references has been added in version 0.28.

clip_into

(1) Signature: *[const]* unsigned int **clip_into** (unsigned int cell, Layout ptr target, const Box box)

Description: Clips the given cell by the given rectangle and produce a new cell with the clip

cell: The cell index of the cell to clip

box: The clip box in database units

target: The target layout

Returns: The index of the new cell in the target layout

This method will cut a rectangular region given by the box from the given cell. The clip will be stored in a new cell in the target layout. The clip will be performed hierarchically. The resulting cell will hold a hierarchy of child cells, which are potentially clipped versions of child cells of the original cell.

Please note that it is important that the database unit of the target layout is identical to the database unit of the source layout to achieve the desired results.This method also assumes that the target layout holds the same layers than the source layout. It will copy shapes to the same layers than they have been on the original layout. This method has been added in version 0.21.

(2) Signature: *[const]* unsigned int **clip_into** (unsigned int cell, Layout ptr target, const DBox box)

Description: Clips the given cell by the given rectangle and produce a new cell with the clip

cell: The cell index of the cell to clip
box: The clip box in micrometer units
target: The target layout
Returns: The index of the new cell in the target layout

This variant which takes a micrometer-unit box has been added in version 0.28.

(3) Signature: *[const]* Cell ptr **clip_into** (const Cell cell, Layout ptr target, const Box box)
Description: Clips the given cell by the given rectangle and produce a new cell with the clip

cell: The reference to the cell to clip
box: The clip box in database units
target: The target layout
Returns: The reference to the new cell in the target layout

This variant which takes cell references instead of cell indexes has been added in version 0.28.

(4) Signature: *[const]* Cell ptr **clip_into** (const Cell cell, Layout ptr target, const DBox box)
Description: Clips the given cell by the given rectangle and produce a new cell with the clip

cell: The reference to the cell to clip
box: The clip box in micrometer units
target: The target layout
Returns: The reference to the new cell in the target layout

This variant which takes a micrometer-unit box and cell references has been added in version 0.28.

convert_cell_to_static

Signature: unsigned int **convert_cell_to_static** (unsigned int cell_index)
Description: Converts a PCell or library cell to a usual (static) cell

Returns: The index of the new cell

This method will create a new cell which contains the static representation of the PCell or library proxy given by "cell_index". If that cell is not a PCell or library proxy, it won't be touched and the input cell index is returned.

This method has been added in version 0.23.

copy_layer

(1) Signature: void **copy_layer** (unsigned int src, unsigned int dest)
Description: Copies a layer

src: The layer index of the source layer.
dest: The layer index of the destination layer.

Copies a layer from the source to the destination layer. The destination layer is not cleared before, so that this method merges shapes from the source with the destination layer.

This method was introduced in version 0.19.

(2) Signature: void **copy_layer** (unsigned int src, unsigned int dest, unsigned int flags)
Description: Copies a layer (selected shape types only)

src: The layer index of the source layer.
dest: The layer index of the destination layer.

flags:	A combination of the shape type flags from Shapes, S... constants

Copies a layer from the source to the destination layer. The destination layer is not cleared before, so that this method merges shapes from the source with the destination layer.

This method variant has been introduced in version 0.28.9.

copy_meta_info

(1) Signature: void **copy_meta_info** (const Layout other)

Description: Copies the meta information from the other layout into this layout

See LayoutMetaInfo for details about cells and meta information. The meta information from this layout will be replaced by the meta information from the other layout.

This method has been introduced in version 0.28.16.

(2) Signature: void **copy_meta_info** (const Layout other, const CellMapping cm)

Description: Copies the meta information from the other layout into this layout for the cells given by the cell mapping

See LayoutMetaInfo for details about cells and meta information. This method will use the source/target cell pairs from the cell mapping object and merge the meta information from each source cell from the 'other' layout into the mapped cell inside self. This method can be used with 'copy_tree_shapes' and similar to copy meta information in addition to the shapes. All cell-specific keys in this layout will be replaced by the respective values from the other layout.

This method has been introduced in version 0.28.16.

copy_tree_shapes

(1) Signature: void **copy_tree_shapes** (const Layout source_layout, const CellMapping cell_mapping)

Description: Copies the shapes for all given mappings in the CellMapping object

source_layout:	The layout where to take the shapes from
cell_mapping:	The cell mapping object that determines how cells are identified between source and target layout

Provide a CellMapping object to specify pairs of cells which are mapped from the source layout to this layout. When constructing such a cell mapping object for example with CellMapping#for_multi_cells_full, use self as the target layout. During the cell mapping construction, the cell mapper will usually create a suitable target hierarchy already. After having completed the cell mapping, use copy_tree_shapes to copy over the shapes from the source to the target layout.

This method has been added in version 0.26.8.

(2) Signature: void **copy_tree_shapes** (const Layout source_layout, const CellMapping cell_mapping, const LayerMapping layer_mapping)

Description: Copies the shapes for all given mappings in the CellMapping object using the given layer mapping

source_layout:	The layout where to take the shapes from
cell_mapping:	The cell mapping object that determines how cells are identified between source and target layout
layer_mapping:	Specifies which layers are copied from the source layout to the target layout

Provide a CellMapping object to specify pairs of cells which are mapped from the source layout to this layout. When constructing such a cell mapping object for example with CellMapping#for_multi_cells_full, use self as the target layout. During the cell mapping construction, the cell mapper will usually create a suitable target hierarchy already. After having completed the cell mapping, use copy_tree_shapes to copy over the shapes from the source to the target layout.

This method has been added in version 0.26.8.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

create_cell

(1) Signature: Cell ptr **create_cell** (string name)

Description: Creates a cell with the given name

name:	The name of the cell to create
Returns:	The Cell object of the newly created cell.

If a cell with that name already exists, the unique name will be chosen for the new cell consisting of the given name plus a suitable suffix.

This method has been introduce in version 0.23 and replaces add_cell.

(2) Signature: Cell ptr **create_cell** (string pcell_name, map<string,variant> params)

Description: Creates a cell as a PCell variant for the PCell with the given name

pcell_name:	The name of the PCell and also the name of the cell to create
params:	The PCell parameters (key/value dictionary)
Returns:	The Cell object of the newly created cell or an existing cell if the PCell has already been used with these parameters.

PCells are instantiated by creating a PCell variant. A PCell variant is linked to the PCell and represents this PCell with a particular parameter set.

This method will look up the PCell by the PCell name and create a new PCell variant for the given parameters. If the PCell has already been instantiated with the same parameters, the original variant will be returned. Hence this method is not strictly creating a cell - only if the required variant has not been created yet.

The parameters are specified as a key/value dictionary with the names being the ones from the PCell declaration.

If no PCell with the given name exists, nil is returned.

This method has been introduce in version 0.24.

(3) Signature: Cell ptr **create_cell** (string name, string lib_name)

Description: Creates a cell with the given name

name:	The name of the library cell and the name of the cell to create
lib_name:	The name of the library where to take the cell from
Returns:	The Cell object of the newly created cell or an existing cell if the library cell has already been used in this layout.

Library cells are imported by creating a 'library proxy'. This is a cell which represents the library cell in the framework of the current layout. The library proxy is linked to the library and will be updated if the library cell is changed.

This method will look up the cell by the given name in the specified library and create a new library proxy for this cell. If the same library cell has already been used, the original library proxy is returned. Hence, strictly speaking this method does not always create a new cell but may return a reference to an existing cell.

If the library name is not valid, nil is returned.

This method has been introduce in version 0.24.

(4) Signature: Cell ptr **create_cell** (string pcell_name, string lib_name, map<string,variant> params)

Description: Creates a cell for a PCell with the given PCell name from the given library

pcell_name:	The name of the PCell and also the name of the cell to create
lib_name:	The name of the library where to take the PCell from
params:	The PCell parameters (key/value dictionary)
Returns:	The Cell object of the newly created cell or an existing cell if this PCell has already been used with the given parameters

This method will look up the PCell by the PCell name in the specified library and create a new PCell variant for the given parameters plus the library proxy. The parameters must be specified as a key/value dictionary with the names being the ones from the PCell declaration.

If no PCell with the given name exists or the library name is not valid, nil is returned. Note that this function - despite the name - may not always create a new cell, but return an existing cell if the PCell from the library has already been used with the given parameters.

This method has been introduce in version 0.24.

dbu

Signature: *[const]* double **dbu**

Description: Gets the database unit

The database unit is the value of one units distance in micrometers. For numerical reasons and to be compliant with the GDS2 format, the database objects use integer coordinates. The basic unit of these coordinates is the database unit. You can convert coordinates to micrometers by multiplying the integer value with the database unit. Typical values for the database unit are 0.001 micrometer (one nanometer).

Python specific notes:
The object exposes a readable attribute 'dbu'. This is the getter.

dbu=

Signature: void **dbu=** (double dbu)

Description: Sets the database unit

See dbu for a description of the database unit.

Python specific notes:
The object exposes a writable attribute 'dbu'. This is the setter.

delete_cell

Signature: void **delete_cell** (unsigned int cell_index)

Description: Deletes a cell

cell_index:	The index of the cell to delete

This deletes a cell but not the sub cells of the cell. These subcells will likely become new top cells unless they are used otherwise. All instances of this cell are deleted as well. Hint: to delete multiple cells, use "delete_cells" which is far more efficient in this case.

This method has been introduced in version 0.20.

delete_cell_rec

Signature: void **delete_cell_rec** (unsigned int cell_index)

Description: Deletes a cell plus all subcells

cell_index:	The index of the cell to delete

This deletes a cell and also all sub cells of the cell. In contrast to prune_cell, all cells are deleted together with their instances even if they are used otherwise.

This method has been introduced in version 0.20.

delete_cells

Signature: void **delete_cells** (unsigned int[] cell_index_list)

Description: Deletes multiple cells

cell_index_list: An array of cell indices of the cells to delete

This deletes the cells but not the sub cells of these cells. These subcells will likely become new top cells unless they are used otherwise. All instances of these cells are deleted as well.

This method has been introduced in version 0.20.

delete_layer

Signature: void **delete_layer** (unsigned int layer_index)

Description: Deletes a layer

layer_index: The index of the layer to delete.

This method frees the memory allocated for the shapes of this layer and remembers the layer's index for reuse when the next layer is allocated.

delete_property

Signature: void **delete_property** (variant key)

Description: Deletes the user property with the given key

This method is a convenience method that deletes the property with the given key. It does nothing if no property with that key exists. Using that method is more convenient than creating a new property set with a new ID and assigning that properties ID. This method may change the properties ID.

This method has been introduced in version 0.24.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Layout ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_cell

Signature: *[iter]* Cell **each_cell**

Description: Iterates the unsorted cell list

each_cell_bottom_up

Signature: *[iter]* unsigned int **each_cell_bottom_up**

Description: Iterates the bottom-up sorted cell list

In bottom-up traversal a cell is not delivered before the last child cell of this cell has been delivered. The bottom-up iterator does not deliver cells but cell indices actually.

each_cell_top_down

Signature: *[iter]* unsigned int **each_cell_top_down**

Description: begin iterator of the top-down sorted cell list

The top-down cell list has the property of delivering all cells before they are instantiated. In addition the first cells are all top cells. There is at least one top cell. The top-down iterator does not deliver cells but cell indices actually.

each_meta_info

Signature: *[const,iter]* LayoutMetaInfo **each_meta_info**

Description: Iterates over the meta information of the layout

See LayoutMetaInfo for details about layouts and meta information.

This method has been introduced in version 0.25.

each_top_cell

Signature: *[iter]* unsigned int **each_top_cell**

Description: Iterates the top cells

A layout may have an arbitrary number of top cells. The usual case however is that there is one top cell.

end_changes

Signature: void **end_changes**

Description: Cancels the "in changes" state (see "start_changes")

error_layer

Signature: *[const]* unsigned int **error_layer**

Description: Returns the index of the error layer

The error layer is used to place error texts on it, for example when a PCell evaluation fails.

This method has been added in version 0.23.13.

find_layer

(1) Signature: variant **find_layer** (const LayerInfo info)

Description: Finds a layer with the given properties

If a layer with the given properties already exists, this method will return the index of that layer.If no such layer exists, it will return nil.

In contrast to layer, this method will also find layers matching by name only. For example:

```
# finds layer '17/0' and 'name (17/0)':
index = layout.find_layer(RBA::LayerInfo::new(17, 0))
# finds layer 'name' (first priority), but also 'name (17/0)' (second
 priority):
index = layout.find_layer(RBA::LayerInfo::new('name'))
# note that this will not match layer 'name (17/0)' and create a new name-
only layer:
index = layout.layer(RBA::LayerInfo::new('name'))
```

This method has been introduced in version 0.23 and has been extended to name queries in version 0.28.11.

(2) Signature: variant **find_layer** (string name)

Description: Finds a layer with the given name

If a layer with the given name already exists, this method will return the index of that layer.If no such layer exists, it will return nil.

In contrast to layer, this method will also find numbered layers if the name matches. For example:

```
# finds layer 'name' (first priority), but also 'name (17/0)' (second
 priority):
index = layout.find_layer('name')
# note that this will not match layer 'name (17/0)' and create a new name-
only layer:
index = layout.layer('name')
```

This method has been introduced in version 0.23 and has been extended to name queries in version 0.28.11.

(3) Signature: variant **find_layer** (int layer, int datatype)

Description: Finds a layer with the given layer and datatype number

If a layer with the given layer/datatype already exists, this method will return the index of that layer.If no such layer exists, it will return nil.

This method has been introduced in version 0.23.

(4) Signature: variant **find_layer** (int layer, int datatype, string name)

Description: Finds a layer with the given layer and datatype number and name

If a layer with the given layer/datatype/name already exists, this method will return the index of that layer.If no such layer exists, it will return nil.

This method has been introduced in version 0.23.

flatten

Signature: void **flatten** (unsigned int cell_index, int levels, bool prune)

Description: Flattens the given cell

cell_index:	The cell which should be flattened
levels:	The number of hierarchy levels to flatten (-1: all, 0: none, 1: one level etc.)
prune:	Set to true to remove orphan cells.

This method propagates all shapes and instances from the specified number of hierarchy levels below into the given cell. It also removes the instances of the cells from which the shapes came from, but does not remove the cells themselves if prune is set to false. If prune is set to true, these cells are removed if not used otherwise.

This method has been introduced in version 0.20.

flatten_into

Signature: void **flatten_into** (unsigned int source_cell_index, unsigned int target_cell_index, const ICplxTrans trans, int levels)

Description: Flattens the given cell into another cell

source_cell_index:	The source cell which should be flattened
target_cell_index:	The target cell into which the resulting objects are written

trans:	The transformation to apply on the output shapes and instances
levels:	The number of hierarchy levels to flatten (-1: all, 0: none, 1: one level etc.)

This method works like 'flatten', but allows specification of a target cell which can be different from the source cell plus a transformation which is applied for all shapes and instances in the target cell.

In contrast to the 'flatten' method, the source cell is not modified.

This method has been introduced in version 0.24.

get_info

Signature: *[const]* LayerInfo **get_info** (unsigned int index)

Description: Gets the info structure for a specified layer

If the layer index is not a valid layer index, an empty LayerProperties object will be returned.

guiding_shape_layer

Signature: *[const]* unsigned int **guiding_shape_layer**

Description: Returns the index of the guiding shape layer

The guiding shape layer is used to store guiding shapes for PCells.

This method has been added in version 0.22.

has_cell?

Signature: bool **has_cell?** (string name)

Description: Returns true if a cell with a given name exists

Returns true, if the layout has a cell with the given name

has_prop_id?

Signature: *[const]* bool **has_prop_id?**

Description: Returns true, if the layout has user properties

This method has been introduced in version 0.24.

insert

(1) Signature: void **insert** (unsigned int cell_index, int layer, const Region region)

Description: Inserts a region into the given cell and layer

If the region is (conceptionally) a flat region, it will be inserted into the cell's shapes list as a flat sequence of polygons. If the region is a deep (hierarchical) region, it will create a subhierarchy below the given cell and its shapes will be put into the respective cells. Suitable subcells will be picked for inserting the shapes. If a hierarchy already exists below the given cell, the algorithm will try to reuse this hierarchy.

This method has been introduced in version 0.26.

(2) Signature: void **insert** (unsigned int cell_index, int layer, const Edges edges)

Description: Inserts an edge collection into the given cell and layer

If the edge collection is (conceptionally) flat, it will be inserted into the cell's shapes list as a flat sequence of edges. If the edge collection is deep (hierarchical), it will create a subhierarchy below the given cell and its edges will be put into the respective cells. Suitable subcells will be picked for inserting the edges. If a hierarchy already exists below the given cell, the algorithm will try to reuse this hierarchy.

This method has been introduced in version 0.26.

(3) Signature: void **insert** (unsigned int cell_index, int layer, const EdgePairs edge_pairs)

Description: Inserts an edge pair collection into the given cell and layer

If the edge pair collection is (conceptionally) flat, it will be inserted into the cell's shapes list as a flat sequence of edge pairs. If the edge pair collection is deep (hierarchical), it will create a subhierarchy below the given cell and its edge pairs will be put into the respective cells. Suitable subcells will be picked for inserting the edge pairs. If a hierarchy already exists below the given cell, the algorithm will try to reuse this hierarchy.

This method has been introduced in version 0.27.

(4) Signature: void **insert** (unsigned int cell_index, int layer, const Texts texts)

Description: Inserts an text collection into the given cell and layer

If the text collection is (conceptionally) flat, it will be inserted into the cell's shapes list as a flat sequence of texts. If the text collection is deep (hierarchical), it will create a subhierarchy below the given cell and its texts will be put into the respective cells. Suitable subcells will be picked for inserting the texts. If a hierarchy already exists below the given cell, the algorithm will try to reuse this hierarchy.

This method has been introduced in version 0.27.

insert_layer

Signature: unsigned int **insert_layer** (const LayerInfo props)

Description: Inserts a new layer with the given properties

Returns: The index of the newly created layer

insert_layer_at

Signature: void **insert_layer_at** (unsigned int index, const LayerInfo props)

Description: Inserts a new layer with the given properties at the given index

This method will associate the given layer info with the given layer index. If a layer with that index already exists, this method will change the properties of the layer with that index. Otherwise a new layer is created.

insert_special_layer

Signature: unsigned int **insert_special_layer** (const LayerInfo props)

Description: Inserts a new special layer with the given properties

Returns: The index of the newly created layer

Special layers can be used to represent objects that should not participate in normal viewing or other related operations. Special layers are not reported as valid layers.

insert_special_layer_at

Signature: void **insert_special_layer_at** (unsigned int index, const LayerInfo props)

Description: Inserts a new special layer with the given properties at the given index

See insert_special_layer for a description of special layers.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_editable?

Signature: *[const]* bool **is_editable?**

Description: Returns a value indicating whether the layout is editable.

Returns: True, if the layout is editable.

If a layout is editable, in general manipulation methods are enabled and some optimizations are disabled (i.e. shape arrays are expanded).

This method has been introduced in version 0.22.

is_free_layer?

Signature: *[const]* bool **is_free_layer?** (unsigned int layer_index)

Description: Returns true, if a layer index is a free (unused) layer index

Returns: true, if this is the case

This method has been introduced in version 0.26.

is_special_layer?

Signature: *[const]* bool **is_special_layer?** (unsigned int layer_index)

Description: Returns true, if a layer index is a special layer index

Returns: true, if this is the case

is_valid_cell_index?

Signature: *[const]* bool **is_valid_cell_index?** (unsigned int cell_index)

Description: Returns true, if a cell index is a valid index

Returns: true, if this is the case

This method has been added in version 0.20.

is_valid_layer?

Signature: *[const]* bool **is_valid_layer?** (unsigned int layer_index)

Description: Returns true, if a layer index is a valid normal layout layer index

Returns: true, if this is the case

layer

(1) Signature: unsigned int **layer**

Description: Creates a new internal layer

This method will create a new internal layer and return the layer index for this layer. The layer does not have any properties attached to it. That means, it is not going to be saved to a layout file unless it is given database properties with set_info.

This method is equivalent to "layer(RBA::LayerInfo::new())".

This method has been introduced in version 0.25.

(2) Signature: unsigned int **layer** (const LayerInfo info)

Description: Finds or creates a layer with the given properties

If a layer with the given properties already exists, this method will return the index of that layer.If no such layer exists, a new one with these properties will be created and its index will be returned. If "info" is anonymous (info.anonymous? is true), a new layer will always be created.

This method has been introduced in version 0.23.

(3) Signature: unsigned int **layer** (string name)

Description: Finds or creates a layer with the given name

If a layer with the given name already exists, this method will return the index of that layer.If no such layer exists, a new one with this name will be created and its index will be returned.

This method has been introduced in version 0.23.

(4) Signature: unsigned int **layer** (int layer, int datatype)

Description: Finds or creates a layer with the given layer and datatype number

If a layer with the given layer/datatype already exists, this method will return the index of that layer.If no such layer exists, a new one with these properties will be created and its index will be returned.

This method has been introduced in version 0.23.

(5) Signature: unsigned int **layer** (int layer, int datatype, string name)

Description: Finds or creates a layer with the given layer and datatype number and name

If a layer with the given layer/datatype/name already exists, this method will return the index of that layer.If no such layer exists, a new one with these properties will be created and its index will be returned.

This method has been introduced in version 0.23.

layer_indexes

Signature: *[const]* unsigned int[] **layer_indexes**

Description: Gets a list of valid layer's indices

This method returns an array with layer indices representing valid layers.

This method has been introduced in version 0.19.

layer_indices

Signature: *[const]* unsigned int[] **layer_indices**

Description: Gets a list of valid layer's indices

Use of this method is deprecated. Use layer_indexes instead

This method returns an array with layer indices representing valid layers.

This method has been introduced in version 0.19.

layer_infos

Signature: *[const]* LayerInfo[] **layer_infos**

Description: Gets a list of valid layer's properties

The method returns an array with layer properties representing valid layers. The sequence and length of this list corresponds to that of layer_indexes.

This method has been introduced in version 0.25.

layers

Signature: *[const]* unsigned int **layers**

Description: Returns the number of layers

The number of layers reports the maximum (plus 1) layer index used so far. Not all of the layers with an index in the range of 0 to layers-1 needs to be a valid layer. These layers can be either valid, special or unused. Use is_valid_layer? and is_special_layer? to test for the first two states.

library

Signature: *[const]* Library ptr **library**

Description: Gets the library this layout lives in or nil if the layout is not part of a library

This attribute has been introduced in version 0.27.5.

merge_meta_info

(1) Signature: void **merge_meta_info** (const Layout other)

Description: Merges the meta information from the other layout into this layout

See LayoutMetaInfo for details about cells and meta information. Existing keys in this layout will be overwritten by the respective values from the other layout. New keys will be added.

This method has been introduced in version 0.28.16.

(2) Signature: void **merge_meta_info** (const Layout other, const CellMapping cm)

Description: Merges the meta information from the other layout into this layout for the cells given by the cell mapping

See LayoutMetaInfo for details about cells and meta information. This method will use the source/target cell pairs from the cell mapping object and merge the meta information from each source cell from the 'other' layout into the mapped cell inside self. This method can be used with 'copy_tree_shapes' and similar to copy meta information in addition to the shapes. Existing cell-specific keys in this layout will be overwritten by the respective values from the other layout. New keys will be added.

This method has been introduced in version 0.28.16.

meta_info

Signature: new LayoutMetaInfo ptr **meta_info** (string name)

Description: Gets the meta information for a given name

See LayoutMetaInfo for details about layouts and meta information.

If no meta information with the given name exists, nil is returned.

This method has been introduced in version 0.28.8.

meta_info_value

Signature: variant **meta_info_value** (string name)

Description: Gets the meta information value for a given name

See LayoutMetaInfo for details about layouts and meta information.

If no meta information with the given name exists, a nil value will be returned. A more generic version that delivers all fields of the meta information is meta_info.

This method has been introduced in version 0.25. Starting with version 0.28.8, the value is of variant type instead of string only.

move_layer

(1) Signature: void **move_layer** (unsigned int src, unsigned int dest)

Description: Moves a layer

src:	The layer index of the source layer.
dest:	The layer index of the destination layer.

Moves a layer from the source to the destination layer. The target is not cleared before, so that this method merges shapes from the source with the destination layer. The source layer is empty after that operation.

This method was introduced in version 0.19.

(2) Signature: void **move_layer** (unsigned int src, unsigned int dest, unsigned int flags)

Description: Moves a layer (selected shape types only)

src:	The layer index of the source layer.
dest:	The layer index of the destination layer.
flags:	A combination of the shape type flags from Shapes, S... constants

Moves a layer from the source to the destination layer. The target is not cleared before, so that this method merges shapes from the source with the destination layer. The copied shapes are removed from the source layer.

This method variant has been introduced in version 0.28.9.

move_tree_shapes

(1) Signature: void **move_tree_shapes** (Layout source_layout, const CellMapping cell_mapping)

Description: Moves the shapes for all given mappings in the CellMapping object

This method acts like the corresponding copy_tree_shapes method, but removes the shapes from the source layout after they have been copied.

This method has been added in version 0.26.8.

(2) Signature: void **move_tree_shapes** (Layout source_layout, const CellMapping cell_mapping, const LayerMapping layer_mapping)

Description: Moves the shapes for all given mappings in the CellMapping object using the given layer mapping

This method acts like the corresponding copy_tree_shapes method, but removes the shapes from the source layout after they have been copied.

This method has been added in version 0.26.8.

multi_clip

(1) Signature: unsigned int[] **multi_clip** (unsigned int cell, Box[] boxes)

Description: Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.

cell:	The cell index of the cell to clip
boxes:	The clip boxes in database units
Returns:	The indexes of the new cells

This method will cut rectangular regions given by the boxes from the given cell. The clips will be stored in a new cells whose indexed are returned. The clips will be performed hierarchically. The resulting cells will hold a hierarchy of child cells, which are potentially clipped versions of child cells of the original cell. This version is somewhat more efficient than doing individual clips because the clip cells may share clipped versions of child cells.

This method has been added in version 0.21.

(2) Signature: unsigned int[] **multi_clip** (unsigned int cell, DBox[] boxes)

Description: Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.

cell:	The cell index of the cell to clip
boxes:	The clip boxes in micrometer units
Returns:	The indexes of the new cells

This variant which takes micrometer-unit boxes has been added in version 0.28.

(3) Signature: Cell ptr[] **multi_clip** (const Cell cell, Box[] boxes)

Description: Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.

cell:	The reference to the cell to clip
boxes:	The clip boxes in database units
Returns:	The references to the new cells

This variant which takes cell references has been added in version 0.28.

(4) Signature: Cell ptr[] **multi_clip** (const Cell cell, DBox[] boxes)

Description: Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.

cell:	The reference to the cell to clip
boxes:	The clip boxes in micrometer units
Returns:	The references to the new cells

This variant which takes cell references and micrometer-unit boxes has been added in version 0.28.

multi_clip_into

(1) Signature: unsigned int[] **multi_clip_into** (unsigned int cell, Layout ptr target, Box[] boxes)

Description: Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.

cell:	The cell index of the cell to clip
boxes:	The clip boxes in database units
target:	The target layout
Returns:	The indexes of the new cells

This method will cut rectangular regions given by the boxes from the given cell. The clips will be stored in a new cells in the given target layout. The clips will be performed hierarchically. The resulting cells will hold a hierarchy of child cells, which are potentially clipped versions of child cells of the original cell. This version is somewhat more efficient than doing individual clips because the clip cells may share clipped versions of child cells.

Please note that it is important that the database unit of the target layout is identical to the database unit of the source layout to achieve the desired results. This method also assumes that the target layout holds the same layers than the source layout. It will copy shapes to the same layers than they have been on the original layout.

This method has been added in version 0.21.

(2) Signature: unsigned int[] **multi_clip_into** (unsigned int cell, Layout ptr target, DBox[] boxes)

Description: Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.

cell:	The cell index of the cell to clip
boxes:	The clip boxes in database units
target:	The target layout
Returns:	The indexes of the new cells

This variant which takes micrometer-unit boxes has been added in version 0.28.

(3) Signature: Cell ptr[] **multi_clip_into** (const Cell cell, Layout ptr target, Box[] boxes)

Description: Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.

cell:	The reference the cell to clip
boxes:	The clip boxes in database units
target:	The target layout
Returns:	The references to the new cells

This variant which takes cell references boxes has been added in version 0.28.

(4) Signature: Cell ptr[] **multi_clip_into** (const Cell cell, Layout ptr target, DBox[] boxes)

Description: Clips the given cell by the given rectangles and produces new cells with the clips, one for each rectangle.

cell:	The reference the cell to clip
boxes:	The clip boxes in micrometer units
target:	The target layout
Returns:	The references to the new cells

This variant which takes cell references and micrometer-unit boxes has been added in version 0.28.

new

(1) Signature: *[static]* new Layout ptr **new** (Manager manager)

Description: Creates a layout object attached to a manager

This constructor specifies a manager object which is used to store undo information for example.

Starting with version 0.25, layouts created with the default constructor are always editable. Before that version, they inherited the editable flag from the application.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Layout ptr **new**

Description: Creates a layout object

Starting with version 0.25, layouts created with the default constructor are always editable. Before that version, they inherited the editable flag from the application.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Layout ptr **new** (bool editable, Manager manager)

Description: Creates a layout object attached to a manager

This constructor specifies a manager object which is used to store undo information for example. It also allows one to specify whether the layout is editable. In editable mode, some optimizations are disabled and the layout can be manipulated through a variety of methods.

This method was introduced in version 0.22.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Layout ptr **new** (bool editable)

Description: Creates a layout object

This constructor specifies whether the layout is editable. In editable mode, some optimizations are disabled and the layout can be manipulated through a variety of methods.

This method was introduced in version 0.22.

Python specific notes:
This method is the default initializer of the object.

pcell_declaration

(1) Signature: *[const]* const PCellDeclaration ptr **pcell_declaration** (string name)

Description: Gets a reference to the PCell declaration for the PCell with the given name

Returns a reference to the local PCell declaration with the given name. If the name is not a valid PCell name, this method returns nil.

Usually this method is used on library layouts that define PCells. Note that this method cannot be used on the layouts using the PCell from a library.

This method has been introduced in version 0.22.

(2) Signature: *[const]* const PCellDeclaration ptr **pcell_declaration** (unsigned int pcell_id)

Description: Gets a reference to the PCell declaration for the PCell with the given PCell ID.

Returns a reference to the local PCell declaration with the given PCell id. If the parameter is not a valid PCell ID, this method returns nil. The PCell ID is the number returned by register_pcell for example.

Usually this method is used on library layouts that define PCells. Note that this method cannot be used on the layouts using the PCell from a library.

This method has been introduced in version 0.22.

pcell_id

Signature: *[const]* unsigned int **pcell_id** (string name)

Description: Gets the ID of the PCell with the given name

This method is equivalent to 'pcell_declaration(name).id'.

This method has been introduced in version 0.22.

pcell_ids

Signature: *[const]* unsigned int[] **pcell_ids**

Description: Gets the IDs of the PCells registered in the layout

Returns an array of PCell IDs.

This method has been introduced in version 0.24.

pcell_names

Signature: *[const]* string[] **pcell_names**

Description: Gets the names of the PCells registered in the layout

Returns an array of PCell names.

This method has been introduced in version 0.24.

prop_id

Signature: *[const]* unsigned long **prop_id**

Description: Gets the properties ID associated with the layout

This method has been introduced in version 0.24.

Python specific notes:
The object exposes a readable attribute 'prop_id'. This is the getter.

prop_id=

Signature: void **prop_id=** (unsigned long id)

Description: Sets the properties ID associated with the layout

This method is provided, if a properties ID has been derived already. Usually it's more convenient to use delete_property, set_property or property.

This method has been introduced in version 0.24.

Python specific notes:
The object exposes a writable attribute 'prop_id'. This is the setter.

properties

Signature: *[const]* variant[] **properties** (unsigned long properties_id)

Description: Gets the properties set for a given properties ID

properties_id:	The properties ID to get the properties for
Returns:	The array of variants (see properties_id)

Basically performs the backward conversion of the 'properties_id' method. Given a properties ID, returns the properties set as an array of pairs of variants. In this array, each key and the value

are stored as pairs (arrays with two elements). If the properties ID is not valid, an empty array is returned.

properties_id

Signature: unsigned long **properties_id** (variant[] properties)

Description: Gets the properties ID for a given properties set

properties:	The array of pairs of variants (both elements can be integer, double or string)
Returns:	The unique properties ID for that set

Before a set of properties can be attached to a shape, it must be converted into an ID that is unique for that set. The properties set must be given as a list of pairs of variants, each pair describing a name and a value. The name acts as the key for the property and does not need to be a string (it can be an integer or double value as well). The backward conversion can be performed with the 'properties' method.

property

Signature: variant **property** (variant key)

Description: Gets the user property with the given key

This method is a convenience method that gets the property with the given key. If no property with that key exists, it will return nil. Using that method is more convenient than using the properties ID to retrieve the property value. This method has been introduced in version 0.24.

prune_cell

Signature: void **prune_cell** (unsigned int cell_index, int levels)

Description: Deletes a cell plus subcells not used otherwise

cell_index:	The index of the cell to delete
levels:	The number of hierarchy levels to consider (-1: all, 0: none, 1: one level etc.)

This deletes a cell and also all sub cells of the cell which are not used otherwise. The number of hierarchy levels to consider can be specified as well. One level of hierarchy means that only the direct children of the cell are deleted with the cell itself. All instances of this cell are deleted as well.

This method has been introduced in version 0.20.

prune_subcells

Signature: void **prune_subcells** (unsigned int cell_index, int levels)

Description: Deletes all sub cells of the cell which are not used otherwise down to the specified level of hierarchy

cell_index:	The root cell from which to delete a sub cells
levels:	The number of hierarchy levels to consider (-1: all, 0: none, 1: one level etc.)

This deletes all sub cells of the cell which are not used otherwise. All instances of the deleted cells are deleted as well. It is possible to specify how many levels of hierarchy below the given root cell are considered.

This method has been introduced in version 0.20.

read

(1) Signature: LayerMap **read** (string filename)

Description: Load the layout from the given file

filename:	The name of the file to load.
Returns:	A layer map that contains the mapping used by the reader including the layers that have been created.

The format of the file is determined automatically and automatic unzipping is provided. No particular options can be specified.

This method has been added in version 0.18.

(2) Signature: LayerMap **read** (string filename, const LoadLayoutOptions options)

Description: Load the layout from the given file with options

filename:	The name of the file to load.
options:	The options object specifying further options for the reader.
Returns:	A layer map that contains the mapping used by the reader including the layers that have been created.

The format of the file is determined automatically and automatic unzipping is provided. In this version, some reader options can be specified.

This method has been added in version 0.18.

refresh

Signature: void **refresh**

Description: Calls Cell#refresh on all cells inside this layout

This method is useful to recompute all PCells from a layout. Note that this does not update PCells which are linked from a library. To recompute PCells from a library, you need to use Library#refresh on the library object from which the PCells are imported.

This method has been introduced in version 0.27.9.

register_pcell

Signature: unsigned int **register_pcell** (string name, PCellDeclaration ptr declaration)

Description: Registers a PCell declaration under the given name

Registers a local PCell in the current layout. If a declaration with that name already exists, it is replaced with the new declaration.

This method has been introduced in version 0.22.

remove_meta_info

Signature: void **remove_meta_info** (string name)

Description: Removes meta information from the layout

See LayoutMetaInfo for details about layouts and meta information. This method has been introduced in version 0.25.

rename_cell

Signature: void **rename_cell** (unsigned int index, string name)

Description: Renames the cell with given index

The cell with the given index is renamed to the given name. NOTE: it is not ensured that the name is unique. This method allows assigning identical names to different cells which usually breaks things. Consider using unique_cell_name to generate truely unique names.

scale_and_snap

(1) Signature: void **scale_and_snap** (Cell cell, int grid, int mult, int div)

Description: Scales and snaps the layout below a given cell by the given rational factor and snaps to the given grid

This method is useful to scale a layout by a non-integer factor. The scale factor is given by the rational number mult / div. After scaling, the layout will be snapped to the given grid.

Snapping happens 'as-if-flat' - that is, touching edges will stay touching, regardless of their hierarchy path. To achieve this, this method usually needs to produce cell variants.

This method has been introduced in version 0.26.1.

(2) Signature: void **scale_and_snap** (unsigned int cell_index, int grid, int mult, int div)

Description: Scales and snaps the layout below a given cell by the given rational factor and snaps to the given grid

Like the other version of scale_and_snap, but taking a cell index for the argument.

This method has been introduced in version 0.26.1.

set_info

Signature: void **set_info** (unsigned int index, const LayerInfo props)

Description: Sets the info structure for a specified layer

set_property

Signature: void **set_property** (variant key, variant value)

Description: Sets the user property with the given key to the given value

This method is a convenience method that sets the property with the given key to the given value. If no property with that key exists, it will create one. Using that method is more convenient than creating a new property set with a new ID and assigning that properties ID. This method may change the properties ID. Note: GDS only supports integer keys. OASIS supports numeric and string keys. This method has been introduced in version 0.24.

start_changes

Signature: void **start_changes**

Description: Signals the start of an operation bringing the layout into invalid state

This method should be called whenever the layout is about to be brought into an invalid state. After calling this method, under_construction? returns true which tells foreign code (i.e. the asynchronous painter or the cell tree view) not to use this layout object.

This state is cancelled by the end_changes method. The start_changes method can be called multiple times and must be cancelled the same number of times.

This method can be used to speed up certain operations. For example iterating over the layout with a RecursiveShapeIterator while modifying other layers of the layout can be very inefficient, because inside the loop the layout's state is invalidate and updated frequently. Putting a update and start_changes sequence before the loop (use both methods in that order!) and a end_changes call after the loop can improve the performance dramatically.

In addition, it can be necessary to prevent redraw operations in certain cases by using start_changes .. end_changes, in particular when it is possible to put a layout object into an invalid state temporarily.

While the layout is under construction update can be called to update the internal state explicitly if required. This for example might be necessary to update the cell bounding boxes or to redo the sorting for region queries.

swap_layers

Signature: void **swap_layers** (unsigned int a, unsigned int b)

Description: Swap two layers

a: The first of the layers to swap.

b: The second of the layers to swap.

Swaps the shapes of both layers.

This method was introduced in version 0.19.

technology

Signature: *[const]* const Technology ptr **technology**

Description: Gets the Technology object of the technology this layout is associated with or nil if the layout is not associated with a technology

This method has been introduced in version 0.27. Before that, the technology has been kept in the 'technology' meta data element.

technology_name

Signature: *[const]* string **technology_name**

Description: Gets the name of the technology this layout is associated with

This method has been introduced in version 0.27. Before that, the technology has been kept in the 'technology' meta data element.

Python specific notes:
The object exposes a readable attribute 'technology_name'. This is the getter.

technology_name=

Signature: void **technology_name=** (string name)

Description: Sets the name of the technology this layout is associated with

Changing the technology name will re-assess all library references because libraries can be technology specified. Cell layouts may be substituted during this re-assessment.

This method has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'technology_name'. This is the setter.

top_cell

Signature: Cell ptr **top_cell**

Description: Returns the top cell object

Returns: The Cell object of the top cell

If the layout has a single top cell, this method returns the top cell's Cell object. If the layout does not have a top cell, this method returns "nil". If the layout has multiple top cells, this method raises an error.

This method has been introduced in version 0.23.

top_cells

Signature: Cell ptr[] **top_cells**

Description: Returns the top cell objects

Returns: The Cell objects of the top cells

This method returns and array of Cell objects representing the top cells of the layout. This array can be empty, if the layout does not have a top cell (i.e. no cell at all).

This method has been introduced in version 0.23.

transform

(1) Signature: void **transform** (const Trans trans)

Description: Transforms the layout with the given transformation

This method has been introduced in version 0.23.

(2) Signature: void **transform** (const ICplxTrans trans)

Description: Transforms the layout with the given complex integer transformation

This method has been introduced in version 0.23.

(3) Signature: void **transform** (const DTrans trans)

Description: Transforms the layout with the given transformation, which is in micrometer units

This variant will internally translate the transformation's displacement into database units. Apart from that, it behaves identical to the version with a Trans argument.

This variant has been introduced in version 0.25.

(4) Signature: void **transform** (const DCplxTrans trans)

Description: Transforms the layout with the given complex integer transformation, which is in micrometer units

This variant will internally translate the transformation's displacement into database units. Apart from that, it behaves identical to the version with a ICplxTrans argument.

This method has been introduced in version 0.23.

under_construction?

Signature: *[const]* bool **under_construction?**

Description: Returns true if the layout object is under construction

A layout object is either under construction if a transaction is ongoing or the layout is brought into invalid state by "start_changes".

unique_cell_name

Signature: *[const]* string **unique_cell_name** (string name)

Description: Creates a new unique cell name from the given name

Returns: A unique name derived from the argument

If a cell with the given name exists, a suffix will be added to make the name unique. Otherwise, the argument will be returned unchanged.

The returned name can be used to rename cells without risk of creating name clashes.

This method has been introduced in version 0.28.

update

Signature: void **update**

Description: Updates the internals of the layout

This method updates the internal state of the layout. Usually this is done automatically This method is provided to ensure this explicitly. This can be useful while using start_changes and end_changes to wrap a performance-critical operation. See start_changes for more details.

write

(1) Signature: void **write** (string filename, bool gzip, const SaveLayoutOptions options)

Description: Writes the layout to a stream file

filename: The file to which to write the layout

gzip: Ignored

options: The option set to use for writing. See SaveLayoutOptions for details

Use of this method is deprecated

Starting with version 0.23, this variant is deprecated since the more convenient variant with two parameters automatically determines the compression mode from the file name. The gzip parameter is ignored staring with version 0.23.

(2) Signature: void **write** (string filename, const SaveLayoutOptions options)

Description: Writes the layout to a stream file

filename: The file to which to write the layout

options: The option set to use for writing. See SaveLayoutOptions for details

This version automatically determines the compression mode from the file name. The file is written with zlib compression if the suffix is ".gz" or ".gzip".

This variant has been introduced in version 0.23.

(3) Signature: void **write** (string filename)

Description: Writes the layout to a stream file

filename: The file to which to write the layout

4.53. API reference - Class SaveLayoutOptions

Notation used in Ruby API documentation

Module: db

Description: Options for saving layouts

This class describes the various options for saving a layout to a stream file (GDS2, OASIS and others). There are: layers to be saved, cell or cells to be saved, scale factor, format, database unit and format specific options.

Usually the default constructor provides a suitable object. Please note, that the format written is "GDS2" by default. Either explicitly set a format using format= or derive the format from the file name using set_format_from_filename.

The layers are specified by either selecting all layers or by defining layer by layer using the add_layer method. select_all_layers will explicitly select all layers for saving, deselect_all_layers will explicitly clear the list of layers.

Cells are selected in a similar fashion: by default, all cells are selected. Using add_cell, specific cells can be selected for saving. All these cells plus their hierarchy will then be written to the stream file.

Public constructors

new SaveLayoutOptions ptr	**new**		Default constructor

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add_cell**	(unsigned int cell_index)	Add a cell (plus hierarchy) to be saved
	void	**add_layer**	(unsigned int layer_index, const LayerInfo properties)	Add a layer to be saved
	void	**add_this_cell**	(unsigned int cell_index)	Adds a cell to be saved
	void	**assign**	(const SaveLayoutOptions other)	Assigns another object to self
	void	**cif_blank_separator=**	(bool flag)	Sets a flag indicating whether blanks shall be used as x/y separator characters

[const]	bool	**cif_blank_separator?**		Gets a flag indicating whether blanks shall be used as x/y separator characters
	void	**cif_dummy_calls=**	(bool flag)	Sets a flag indicating whether dummy calls shall be written
[const]	bool	**cif_dummy_calls?**		Gets a flag indicating whether dummy calls shall be written
	void	**clear_cells**		Clears all cells to be saved
[const]	double	**dbu**		Get the explicit database unit if one is set
	void	**dbu=**	(double dbu)	Set the database unit to be used in the stream file
	void	**deselect_all_layers**		Unselect all layers: no layer will be saved
[const]	new SaveLayoutOptions ptr	**dup**		Creates a copy of self
[const]	int	**dxf_polygon_mode**		Specifies how to write polygons.
	void	**dxf_polygon_mode=**	(int mode)	Specifies how to write polygons.
[const]	string	**format**		Gets the format name
	void	**format=**	(string format)	Select a format
[const]	string	**gds2_libname**		Get the library name
	void	**gds2_libname=**	(string libname)	Set the library name
[const]	unsigned int	**gds2_max_cellname_length**		Get the maximum length of cell names
	void	**gds2_max_cellname_length=**	(unsigned int length)	Maximum length of cell names
[const]	unsigned int	**gds2_max_vertex_count**		Gets the maximum number of vertices for polygons to write
	void	**gds2_max_vertex_count=**	(unsigned int count)	Sets the maximum number of vertices for polygons to write
	void	**gds2_multi_xy_records=**	(bool flag)	Uses multiple XY records in BOUNDARY elements for unlimited large polygons
[const]	bool	**gds2_multi_xy_records?**		Gets the property enabling multiple XY records for BOUNDARY elements
	void	**gds2_no_zero_length_paths=**	(bool flag)	Eliminates zero-length paths if true
[const]	bool	**gds2_no_zero_length_paths**		Gets a value indicating whether zero-length paths are eliminated
	void	**gds2_resolve_skew_arrays=**	(bool flag)	Resolves skew arrays into single instances

[const]	bool	**gds2_resolve_skew_arrays?**		Gets a value indicating whether to resolve skew arrays into single instances
[const]	double	**gds2_user_units**		Get the user units
	void	**gds2_user_units=**	(double uu)	Set the users units to write into the GDS file
	void	**gds2_write_cell_properties=**	(bool flag)	Enables writing of cell properties if set to true
[const]	bool	**gds2_write_cell_properties?**		Gets a value indicating whether cell properties are written
	void	**gds2_write_file_properties=**	(bool flag)	Enables writing of file properties if set to true
[const]	bool	**gds2_write_file_properties?**		Gets a value indicating whether layout properties are written
	void	**gds2_write_timestamps=**	(bool flag)	Writes the current time into the GDS2 timestamps if set to true
[const]	bool	**gds2_write_timestamps?**		Gets a value indicating whether the current time is written into the GDS2 timestamp fields
	void	**keep_instances=**	(bool flag)	Enables or disables instances for dropped cells
[const]	bool	**keep_instances?**		Gets a flag indicating whether instances will be kept even if the target cell is dropped
[const]	double	**mag_lambda**		Gets the lambda value
	void	**mag_lambda=**	(double lambda)	Specifies the lambda value to used for writing
[const]	string	**mag_tech**		Gets the technology string used for writing
	void	**mag_tech=**	(string tech)	Specifies the technology string used for writing
	void	**mag_write_timestamp=**	(bool f)	Specifies whether to write a timestamp
[const]	bool	**mag_write_timestamp?**		Gets a value indicating whether to write a timestamp
	void	**no_empty_cells=**	(bool flag)	Don't write empty cells if this flag is set
[const]	bool	**no_empty_cells?**		Returns a flag indicating whether empty cells are not written.
[const]	int	**oasis_compression_level**		Get the OASIS compression level
	void	**oasis_compression_level=**	(int level)	Set the OASIS compression level
	void	**oasis_permissive=**	(bool flag)	Sets OASIS permissive mode
[const]	bool	**oasis_permissive?**		Gets the OASIS permissive mode
	void	**oasis_recompress=**	(bool flag)	Sets OASIS recompression mode

[const]	bool	**oasis_recompress?**		Gets the OASIS recompression mode
	void	**oasis_strict_mode=**	(bool flag)	Sets a value indicating whether to write strict-mode OASIS files
[const]	bool	**oasis_strict_mode?**		Gets a value indicating whether to write strict-mode OASIS files
[const]	string	**oasis_substitution_char**		Gets the substitution character
	void	**oasis_substitution_char=**	(string char)	Sets the substitution character for a-strings and n-strings
	void	**oasis_write_cblocks=**	(bool flag)	Sets a value indicating whether to write compressed CBLOCKS per cell
[const]	bool	**oasis_write_cblocks?**		Gets a value indicating whether to write compressed CBLOCKS per cell
	void	**oasis_write_cell_bounding_boxes=**	(bool flag)	Sets a value indicating whether cell bounding boxes are written
[const]	bool	**oasis_write_cell_bounding_b**		Gets a value indicating whether cell bounding boxes are written
	void	**oasis_write_std_properties=**	(bool flag)	Sets a value indicating whether standard properties will be written
[const]	bool	**oasis_write_std_properties?**		Gets a value indicating whether standard properties will be written
[const]	double	**scale_factor**		Gets the scaling factor currently set
	void	**scale_factor=**	(double scale_factor)	Set the scaling factor for the saving
	void	**select_all_cells**		Select all cells to save
	void	**select_all_layers**		Select all layers to be saved
	void	**select_cell**	(unsigned int cell_index)	Selects a cell to be saved (plus hierarchy below)
	void	**select_this_cell**	(unsigned int cell_index)	Selects a cell to be saved
	bool	**set_format_from_filename**	(string filename)	Select a format from the given file name
	void	**write_context_info=**	(bool flag)	Enables or disables context information
[const]	bool	**write_context_info?**		Gets a flag indicating whether context information will be stored

Deprecated methods (protected, public, static, non-static and constructors)

[const]	bool	**cif_blank_separator**	Use of this method is deprecated. Use cif_blank_separator? instead
[const]	bool	**cif_dummy_calls**	Use of this method is deprecated. Use cif_dummy_calls? instead
	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**gds2_no_zero_length_paths**	Use of this method is deprecated. Use gds2_no_zero_length_paths? instead
[const]	bool	**gds2_write_cell_properties**	Use of this method is deprecated. Use gds2_write_cell_properties? instead
[const]	bool	**gds2_write_file_properties**	Use of this method is deprecated. Use gds2_write_file_properties? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_cell

Signature: void **add_cell** (unsigned int cell_index)

Description: Add a cell (plus hierarchy) to be saved

The index of the cell must be a valid index in the context of the layout that will be saved. This method clears the 'select all cells' flag.

This method also implicitly adds the children of that cell. A method that does not add the children in add_this_cell.

add_layer

Signature: void **add_layer** (unsigned int layer_index, const LayerInfo properties)

Description: Add a layer to be saved

Adds the layer with the given index to the layer list that will be written. If all layers have been selected previously, all layers will be unselected first and only the new layer remains.

The 'properties' argument can be used to assign different layer properties than the ones present in the layout. Pass a default LayerInfo object to this argument to use the properties from the layout object. Construct a valid LayerInfo object with explicit layer, datatype and possibly a name to override the properties stored in the layout.

add_this_cell

Signature: void **add_this_cell** (unsigned int cell_index)

Description: Adds a cell to be saved

The index of the cell must be a valid index in the context of the layout that will be saved. This method clears the 'select all cells' flag. Unlike add_cell, this method does not implicitly add all children of that cell.

This method has been added in version 0.23.

assign

Signature: void **assign** (const SaveLayoutOptions other)

Description: Assigns another object to self

cif_blank_separator

Signature: *[const]* bool **cif_blank_separator**

Description: Gets a flag indicating whether blanks shall be used as x/y separator characters

Use of this method is deprecated. Use cif_blank_separator? instead

See cif_blank_separator= method for a description of that property. This property has been added in version 0.23.10.

The predicate version (cif_blank_separator?) has been added in version 0.25.1.

Python specific notes:
The object exposes a readable attribute 'cif_blank_separator'. This is the getter.
This method is available as 'cif_blank_separator_' in Python to distiguish it from the property with the same name.

cif_blank_separator=

Signature: void **cif_blank_separator=** (bool flag)

Description: Sets a flag indicating whether blanks shall be used as x/y separator characters

If this property is set to true, the x and y coordinates are separated with blank characters rather than comma characters. This property has been added in version 0.23.10.

Python specific notes:
The object exposes a writable attribute 'cif_blank_separator'. This is the setter.

cif_blank_separator?

Signature: *[const]* bool **cif_blank_separator?**

Description: Gets a flag indicating whether blanks shall be used as x/y separator characters

See cif_blank_separator= method for a description of that property. This property has been added in version 0.23.10.

The predicate version (cif_blank_separator?) has been added in version 0.25.1.

Python specific notes:
The object exposes a readable attribute 'cif_blank_separator'. This is the getter.
This method is available as 'cif_blank_separator_' in Python to distiguish it from the property with the same name.

cif_dummy_calls

Signature: *[const]* bool **cif_dummy_calls**

Description: Gets a flag indicating whether dummy calls shall be written

Use of this method is deprecated. Use cif_dummy_calls? instead

See cif_dummy_calls= method for a description of that property. This property has been added in version 0.23.10.

The predicate version (cif_blank_separator?) has been added in version 0.25.1.

Python specific notes:
The object exposes a readable attribute 'cif_dummy_calls'. This is the getter.
This method is available as 'cif_dummy_calls_' in Python to distiguish it from the property with the same name.

cif_dummy_calls=

Signature: void **cif_dummy_calls=** (bool flag)

Description: Sets a flag indicating whether dummy calls shall be written

If this property is set to true, dummy calls will be written in the top level entity of the CIF file calling every top cell. This option is useful for enhanced compatibility with other tools.

This property has been added in version 0.23.10.

Python specific notes:
The object exposes a writable attribute 'cif_dummy_calls'. This is the setter.

cif_dummy_calls?

Signature: *[const]* bool **cif_dummy_calls?**

Description: Gets a flag indicating whether dummy calls shall be written

See cif_dummy_calls= method for a description of that property. This property has been added in version 0.23.10.

The predicate version (cif_blank_separator?) has been added in version 0.25.1.

Python specific notes:
The object exposes a readable attribute 'cif_dummy_calls'. This is the getter.
This method is available as 'cif_dummy_calls_' in Python to distiguish it from the property with the same name.

clear_cells

Signature: void **clear_cells**

Description: Clears all cells to be saved

This method can be used to ensure that no cell is selected before add_cell is called to specify a cell. This method clears the 'select all cells' flag.

This method has been added in version 0.22.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

dbu

Signature: *[const]* double **dbu**

Description: Get the explicit database unit if one is set

See dbu= for a description of that attribute.

Python specific notes:
The object exposes a readable attribute 'dbu'. This is the getter.

dbu=

Signature: void **dbu=** (double dbu)

Description: Set the database unit to be used in the stream file

By default, the database unit of the layout is used. This method allows one to explicitly use a different database unit. A scale factor is introduced automatically which scales all layout objects accordingly so their physical dimensions remain the same. When scaling to a larger database unit or one that is not an integer fraction of the original one, rounding errors may occur and the layout may become slightly distorted.

Python specific notes:
The object exposes a writable attribute 'dbu'. This is the setter.

deselect_all_layers

Signature: void **deselect_all_layers**

Description: Unselect all layers: no layer will be saved

This method will clear all layers selected with add_layer so far and clear the 'select all layers' flag. Using this method is the only way to save a layout without any layers.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new SaveLayoutOptions ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

dxf_polygon_mode

Signature: *[const]* int **dxf_polygon_mode**

Description: Specifies how to write polygons.

See dxf_polygon_mode= for a description of this property.

This property has been added in version 0.21.3.

Python specific notes:
The object exposes a readable attribute 'dxf_polygon_mode'. This is the getter.

dxf_polygon_mode=

Signature: void **dxf_polygon_mode=** (int mode)

Description: Specifies how to write polygons.

The mode is 0 (write POLYLINE entities), 1 (write LWPOLYLINE entities), 2 (decompose into SOLID entities), 3 (write HATCH entities), or 4 (write LINE entities).

This property has been added in version 0.21.3. '4', in version 0.25.6.

Python specific notes:
The object exposes a writable attribute 'dxf_polygon_mode'. This is the setter.

format

Signature: *[const]* string **format**

Description: Gets the format name

See format= for a description of that method.

Python specific notes:
The object exposes a readable attribute 'format'. This is the getter.

format=

Signature: void **format=** (string format)

Description: Select a format

The format string can be either "GDS2", "OASIS", "CIF" or "DXF". Other formats may be available if a suitable plugin is installed.

Python specific notes:
The object exposes a writable attribute 'format'. This is the setter.

gds2_libname

Signature: *[const]* string **gds2_libname**

Description: Get the library name

See gds2_libname= method for a description of the library name. This property has been added in version 0.18.

Python specific notes:
The object exposes a readable attribute 'gds2_libname'. This is the getter.

gds2_libname=

Signature: void **gds2_libname=** (string libname)

Description: Set the library name

The library name is the string written into the LIBNAME records of the GDS file. The library name should not be an empty string and is subject to certain limitations in the character choice.

This property has been added in version 0.18.

Python specific notes:
The object exposes a writable attribute 'gds2_libname'. This is the setter.

gds2_max_cellname_length

Signature: *[const]* unsigned int **gds2_max_cellname_length**

Description: Get the maximum length of cell names

See gds2_max_cellname_length= method for a description of the maximum cell name length. This property has been added in version 0.18.

Python specific notes:
The object exposes a readable attribute 'gds2_max_cellname_length'. This is the getter.

gds2_max_cellname_length=

Signature: void **gds2_max_cellname_length=** (unsigned int length)

Description: Maximum length of cell names

This property describes the maximum number of characters for cell names. Longer cell names will be shortened.

This property has been added in version 0.18.

Python specific notes:
The object exposes a writable attribute 'gds2_max_cellname_length'. This is the setter.

gds2_max_vertex_count

Signature: *[const]* unsigned int **gds2_max_vertex_count**

Description: Gets the maximum number of vertices for polygons to write

See gds2_max_vertex_count= method for a description of the maximum vertex count. This property has been added in version 0.18.

Python specific notes:
The object exposes a readable attribute 'gds2_max_vertex_count'. This is the getter.

gds2_max_vertex_count=

Signature: void **gds2_max_vertex_count=** (unsigned int count)

Description: Sets the maximum number of vertices for polygons to write

This property describes the maximum number of point for polygons in GDS2 files. Polygons with more points will be split. The minimum value for this property is 4. The maximum allowed value is about 4000 or 8000, depending on the GDS2 interpretation. If gds2_multi_xy_records is true, this property is not used. Instead, the number of points is unlimited.

This property has been added in version 0.18.

Python specific notes:
The object exposes a writable attribute 'gds2_max_vertex_count'. This is the setter.

gds2_multi_xy_records=

Signature: void **gds2_multi_xy_records=** (bool flag)

Description: Uses multiple XY records in BOUNDARY elements for unlimited large polygons

Setting this property to true allows producing polygons with an unlimited number of points at the cost of incompatible formats. Setting it to true disables the gds2_max_vertex_count setting.

This property has been added in version 0.18.

Python specific notes:
The object exposes a writable attribute 'gds2_multi_xy_records'. This is the setter.

gds2_multi_xy_records?

Signature: *[const]* bool **gds2_multi_xy_records?**

Description: Gets the property enabling multiple XY records for BOUNDARY elements

See gds2_multi_xy_records= method for a description of this property. This property has been added in version 0.18.

Python specific notes:
The object exposes a readable attribute 'gds2_multi_xy_records'. This is the getter.

gds2_no_zero_length_paths

Signature: *[const]* bool **gds2_no_zero_length_paths**

Description: Gets a value indicating whether zero-length paths are eliminated

Use of this method is deprecated. Use gds2_no_zero_length_paths? instead

This property has been added in version 0.23.

Python specific notes:
The object exposes a readable attribute 'gds2_no_zero_length_paths'. This is the getter.
This method is available as 'gds2_no_zero_length_paths_' in Python to distiguish it from the property with the same name.

gds2_no_zero_length_paths=

Signature: void **gds2_no_zero_length_paths=** (bool flag)

Description: Eliminates zero-length paths if true

If this property is set to true, paths with zero length will be converted to BOUNDARY objects.

This property has been added in version 0.23.

Python specific notes:
The object exposes a writable attribute 'gds2_no_zero_length_paths'. This is the setter.

gds2_no_zero_length_paths?

Signature: *[const]* bool **gds2_no_zero_length_paths?**

Description: Gets a value indicating whether zero-length paths are eliminated

This property has been added in version 0.23.

Python specific notes:
The object exposes a readable attribute 'gds2_no_zero_length_paths'. This is the getter.
This method is available as 'gds2_no_zero_length_paths_' in Python to distiguish it from the property with the same name.

gds2_resolve_skew_arrays=

Signature: void **gds2_resolve_skew_arrays=** (bool flag)

Description: Resolves skew arrays into single instances

Setting this property to true will make skew (non-orthogonal) arrays being resolved into single instances. Skew arrays happen if either the row or column vector isn't parallel to x or y axis. Such arrays can cause problems with some legacy software and can be disabled with this option.

This property has been added in version 0.27.1.

Python specific notes:
The object exposes a writable attribute 'gds2_resolve_skew_arrays'. This is the setter.

gds2_resolve_skew_arrays?

Signature: *[const]* bool **gds2_resolve_skew_arrays?**

Description: Gets a value indicating whether to resolve skew arrays into single instances

See gds2_resolve_skew_arrays= method for a description of this property. This property has been added in version 0.27.1.

Python specific notes:
The object exposes a readable attribute 'gds2_resolve_skew_arrays'. This is the getter.

gds2_user_units

Signature: *[const]* double **gds2_user_units**

Description: Get the user units

See gds2_user_units= method for a description of the user units. This property has been added in version 0.18.

Python specific notes:
The object exposes a readable attribute 'gds2_user_units'. This is the getter.

gds2_user_units=

Signature: void **gds2_user_units=** (double uu)

Description: Set the users units to write into the GDS file

The user units of a GDS file are rarely used and usually are set to 1 (micron). The intention of the user units is to specify the display units. KLayout ignores the user unit and uses microns as the display unit. The user unit must be larger than zero.

This property has been added in version 0.18.

Python specific notes:
The object exposes a writable attribute 'gds2_user_units'. This is the setter.

gds2_write_cell_properties

Signature: *[const]* bool **gds2_write_cell_properties**

Description: Gets a value indicating whether cell properties are written

Use of this method is deprecated. Use gds2_write_cell_properties? instead

This property has been added in version 0.23.

Python specific notes:
The object exposes a readable attribute 'gds2_write_cell_properties'. This is the getter.
This method is available as 'gds2_write_cell_properties_' in Python to distiguish it from the property with the same name.

gds2_write_cell_properties=

Signature: void **gds2_write_cell_properties=** (bool flag)

Description: Enables writing of cell properties if set to true

If this property is set to true, cell properties will be written as PROPATTR/PROPVALUE records immediately following the BGNSTR records. This is a non-standard extension and is therefore disabled by default.

This property has been added in version 0.23.

Python specific notes:
The object exposes a writable attribute 'gds2_write_cell_properties'. This is the setter.

gds2_write_cell_properties?

Signature: *[const]* bool **gds2_write_cell_properties?**

Description: Gets a value indicating whether cell properties are written

This property has been added in version 0.23.

Python specific notes:
The object exposes a readable attribute 'gds2_write_cell_properties'. This is the getter.

This method is available as 'gds2_write_cell_properties_' in Python to distiguish it from the property with the same name.

gds2_write_file_properties

Signature: *[const]* bool **gds2_write_file_properties**

Description: Gets a value indicating whether layout properties are written

Use of this method is deprecated. Use gds2_write_file_properties? instead

This property has been added in version 0.24.

Python specific notes:
The object exposes a readable attribute 'gds2_write_file_properties'. This is the getter.
This method is available as 'gds2_write_file_properties_' in Python to distiguish it from the property with the same name.

gds2_write_file_properties=

Signature: void **gds2_write_file_properties=** (bool flag)

Description: Enables writing of file properties if set to true

If this property is set to true, layout properties will be written as PROPATTR/PROPVALUE records immediately following the BGNLIB records. This is a non-standard extension and is therefore disabled by default.

This property has been added in version 0.24.

Python specific notes:
The object exposes a writable attribute 'gds2_write_file_properties'. This is the setter.

gds2_write_file_properties?

Signature: *[const]* bool **gds2_write_file_properties?**

Description: Gets a value indicating whether layout properties are written

This property has been added in version 0.24.

Python specific notes:
The object exposes a readable attribute 'gds2_write_file_properties'. This is the getter.
This method is available as 'gds2_write_file_properties_' in Python to distiguish it from the property with the same name.

gds2_write_timestamps=

Signature: void **gds2_write_timestamps=** (bool flag)

Description: Writes the current time into the GDS2 timestamps if set to true

If this property is set to false, the time fields will all be zero. This somewhat simplifies compare and diff applications.

This property has been added in version 0.21.16.

Python specific notes:
The object exposes a writable attribute 'gds2_write_timestamps'. This is the setter.

gds2_write_timestamps?

Signature: *[const]* bool **gds2_write_timestamps?**

Description: Gets a value indicating whether the current time is written into the GDS2 timestamp fields

This property has been added in version 0.21.16.

Python specific notes:
The object exposes a readable attribute 'gds2_write_timestamps'. This is the getter.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

keep_instances=

Signature: void **keep_instances=** (bool flag)

Description: Enables or disables instances for dropped cells

If this flag is set to true, instances for cells will be written, even if the cell is dropped. That may happen, if cells are selected with select_this_cell or add_this_cell or no_empty_cells is used. Even if cells called by such cells are not selected, instances will be written for that cell if "keep_instances" is true. That feature is supported by the GDS format currently and results in "ghost cells" which have instances but no cell definition.

The default value is false (instances of dropped cells are not written).

This method was introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'keep_instances'. This is the setter.

keep_instances?

Signature: *[const]* bool **keep_instances?**

Description: Gets a flag indicating whether instances will be kept even if the target cell is dropped

See keep_instances= for details about this flag.

This method was introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'keep_instances'. This is the getter.

mag_lambda

Signature: *[const]* double **mag_lambda**

Description: Gets the lambda value

See mag_lambda= method for a description of this attribute. This property has been added in version 0.26.2.

Python specific notes:
The object exposes a readable attribute 'mag_lambda'. This is the getter.

mag_lambda=

Signature: void **mag_lambda=** (double lambda)

Description: Specifies the lambda value to used for writing

The lambda value is the basic unit of the layout. The layout is brought to units of this value. If the layout is not on-grid on this unit, snapping will happen. If the value is less or equal to zero, KLayout will use the lambda value stored inside the layout set by a previous read operation of a MAGIC file. The lambda value is stored in the Layout object as the "lambda" metadata attribute.

This property has been added in version 0.26.2.

Python specific notes:
The object exposes a writable attribute 'mag_lambda'. This is the setter.

mag_tech

Signature: *[const]* string **mag_tech**

Description: Gets the technology string used for writing

See mag_tech= method for a description of this attribute. This property has been added in version 0.26.2.

Python specific notes:
The object exposes a readable attribute 'mag_tech'. This is the getter.

mag_tech=

Signature: void **mag_tech=** (string tech)

Description: Specifies the technology string used for writing

If this string is empty, the writer will try to obtain the technology from the "technology" metadata attribute of the layout.

This property has been added in version 0.26.2.

Python specific notes:
The object exposes a writable attribute 'mag_tech'. This is the setter.

mag_write_timestamp=

Signature: void **mag_write_timestamp=** (bool f)

Description: Specifies whether to write a timestamp

If this attribute is set to false, the timestamp written is 0. This is not permitted in the strict sense, but simplifies comparison of Magic files.

This property has been added in version 0.26.2.

Python specific notes:
The object exposes a writable attribute 'mag_write_timestamp'. This is the setter.

mag_write_timestamp?

Signature: *[const]* bool **mag_write_timestamp?**

Description: Gets a value indicating whether to write a timestamp

See write_timestamp= method for a description of this attribute.

This property has been added in version 0.26.2.

Python specific notes:
The object exposes a readable attribute 'mag_write_timestamp'. This is the getter.

new

Signature: *[static]* new SaveLayoutOptions ptr **new**

Description: Default constructor

This will initialize the scale factor to 1.0, the database unit is set to "same as original" and all layers are selected as well as all cells. The default format is GDS2.

Python specific notes:
This method is the default initializer of the object.

no_empty_cells=

Signature: void **no_empty_cells=** (bool flag)

Description: Don't write empty cells if this flag is set

By default, all cells are written (no_empty_cells is false). This applies to empty cells which do not contain shapes for the specified layers as well as cells which are empty because they reference empty cells only.

Python specific notes:
The object exposes a writable attribute 'no_empty_cells'. This is the setter.

no_empty_cells?

Signature: *[const]* bool **no_empty_cells?**

Description: Returns a flag indicating whether empty cells are not written.

Python specific notes:
The object exposes a readable attribute 'no_empty_cells'. This is the getter.

oasis_compression_level

Signature: *[const]* int **oasis_compression_level**

Description: Get the OASIS compression level

See oasis_compression_level= method for a description of the OASIS compression level.

Python specific notes:
The object exposes a readable attribute 'oasis_compression_level'. This is the getter.

oasis_compression_level=

Signature: void **oasis_compression_level=** (int level)

Description: Set the OASIS compression level

The OASIS compression level is an integer number between 0 and 10. 0 basically is no compression, 1 produces shape arrays in a simple fashion. 2 and higher compression levels will use a more elaborate algorithm to find shape arrays which uses 2nd and further neighbor distances. The higher the level, the higher the memory requirements and run times.

Python specific notes:
The object exposes a writable attribute 'oasis_compression_level'. This is the setter.

oasis_permissive=

Signature: void **oasis_permissive=** (bool flag)

Description: Sets OASIS permissive mode

If this flag is true, certain shapes which cannot be written to OASIS are reported as warnings, not as errors. For example, paths with odd width (are rounded) or polygons with less than three points (are skipped).

This method has been introduced in version 0.25.1.

Python specific notes:
The object exposes a writable attribute 'oasis_permissive'. This is the setter.

oasis_permissive?

Signature: *[const]* bool **oasis_permissive?**

Description: Gets the OASIS permissive mode

See oasis_permissive= method for a description of this predicate. This method has been introduced in version 0.25.1.

Python specific notes:
The object exposes a readable attribute 'oasis_permissive'. This is the getter.

oasis_recompress=

Signature: void **oasis_recompress=** (bool flag)

Description: Sets OASIS recompression mode

If this flag is true, shape arrays already existing will be resolved and compression is applied to the individual shapes again. If this flag is false (the default), shape arrays already existing will be written as such.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'oasis_recompress'. This is the setter.

oasis_recompress?

Signature: *[const]* bool **oasis_recompress?**

Description: Gets the OASIS recompression mode

See oasis_recompress= method for a description of this predicate. This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'oasis_recompress'. This is the getter.

oasis_strict_mode=

Signature: void **oasis_strict_mode=** (bool flag)

Description: Sets a value indicating whether to write strict-mode OASIS files

Setting this property clears all format specific options for other formats such as GDS.

Python specific notes:
The object exposes a writable attribute 'oasis_strict_mode'. This is the setter.

oasis_strict_mode?

Signature: *[const]* bool **oasis_strict_mode?**

Description: Gets a value indicating whether to write strict-mode OASIS files

Python specific notes:
The object exposes a readable attribute 'oasis_strict_mode'. This is the getter.

oasis_substitution_char

Signature: *[const]* string **oasis_substitution_char**

Description: Gets the substitution character

See oasis_substitution_char for details. This attribute has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'oasis_substitution_char'. This is the getter.

oasis_substitution_char=

Signature: void **oasis_substitution_char=** (string char)

Description: Sets the substitution character for a-strings and n-strings

The substitution character is used in place of invalid characters. The value of this attribute is a string which is either empty or a single character. If the string is empty, no substitution is made at the risk of producing invalid OASIS files.

This attribute has been introduce in version 0.23.

Python specific notes:
The object exposes a writable attribute 'oasis_substitution_char'. This is the setter.

oasis_write_cblocks=

Signature: void **oasis_write_cblocks=** (bool flag)

Description: Sets a value indicating whether to write compressed CBLOCKS per cell

Setting this property clears all format specific options for other formats such as GDS.

Python specific notes:
The object exposes a writable attribute 'oasis_write_cblocks'. This is the setter.

oasis_write_cblocks?

Signature: *[const]* bool **oasis_write_cblocks?**

Description: Gets a value indicating whether to write compressed CBLOCKS per cell

Python specific notes:
The object exposes a readable attribute 'oasis_write_cblocks'. This is the getter.

oasis_write_cell_bounding_boxes=

Signature: void **oasis_write_cell_bounding_boxes=** (bool flag)

Description: Sets a value indicating whether cell bounding boxes are written

If this value is set to true, cell bounding boxes are written (S_BOUNDING_BOX). The S_BOUNDING_BOX properties will be attached to the CELLNAME records.

Setting this value to true will also enable writing of other standard properties like S_TOP_CELL (see oasis_write_std_properties=). By default, cell bounding boxes are not written, but standard properties are.

This method has been introduced in version 0.24.3.

Python specific notes:
The object exposes a writable attribute 'oasis_write_cell_bounding_boxes'. This is the setter.

oasis_write_cell_bounding_bo

Signature: *[const]* bool **oasis_write_cell_bounding_boxes?**

Description: Gets a value indicating whether cell bounding boxes are written

	See oasis_write_cell_bounding_boxes= method for a description of this flag. This method has been introduced in version 0.24.3. **Python specific notes:** The object exposes a readable attribute 'oasis_write_cell_bounding_boxes'. This is the getter.
oasis_write_std_properties=	**Signature**: void **oasis_write_std_properties=** (bool flag) **Description**: Sets a value indicating whether standard properties will be written If this value is false, no standard properties are written. If true, S_TOP_CELL and some other global standard properties are written. In addition, oasis_write_cell_bounding_boxes= can be used to write cell bounding boxes using S_BOUNDING_BOX. By default, this flag is true and standard properties are written. Setting this property to false clears the oasis_write_cell_bounding_boxes flag too. This method has been introduced in version 0.24. **Python specific notes:** The object exposes a writable attribute 'oasis_write_std_properties'. This is the setter.
oasis_write_std_properties?	**Signature**: *[const]* bool **oasis_write_std_properties?** **Description**: Gets a value indicating whether standard properties will be written See oasis_write_std_properties= method for a description of this flag. This method has been introduced in version 0.24. **Python specific notes:** The object exposes a readable attribute 'oasis_write_std_properties'. This is the getter.
scale_factor	**Signature**: *[const]* double **scale_factor** **Description**: Gets the scaling factor currently set **Python specific notes:** The object exposes a readable attribute 'scale_factor'. This is the getter.
scale_factor=	**Signature**: void **scale_factor=** (double scale_factor) **Description**: Set the scaling factor for the saving Using a scaling factor will scale all objects accordingly. This scale factor adds to a potential scaling implied by using an explicit database unit. Be aware that rounding effects may occur if fractional scaling factors are used. By default, no scaling is applied. **Python specific notes:** The object exposes a writable attribute 'scale_factor'. This is the setter.
select_all_cells	**Signature**: void **select_all_cells** **Description**: Select all cells to save This method will clear all cells specified with add_cells so far and set the 'select all cells' flag. This is the default.
select_all_layers	**Signature**: void **select_all_layers** **Description**: Select all layers to be saved This method will clear all layers selected with add_layer so far and set the 'select all layers' flag. This is the default.

select_cell

Signature: void **select_cell** (unsigned int cell_index)

Description: Selects a cell to be saved (plus hierarchy below)

This method is basically a convenience method that combines clear_cells and add_cell. This method clears the 'select all cells' flag.

This method has been added in version 0.22.

select_this_cell

Signature: void **select_this_cell** (unsigned int cell_index)

Description: Selects a cell to be saved

This method is basically a convenience method that combines clear_cells and add_this_cell. This method clears the 'select all cells' flag.

This method has been added in version 0.23.

set_format_from_filename

Signature: bool **set_format_from_filename** (string filename)

Description: Select a format from the given file name

This method will set the format according to the file's extension.

This method has been introduced in version 0.22. Beginning with version 0.23, this method always returns true, since the only consumer for the return value, Layout#write, now ignores that parameter and automatically determines the compression mode from the file name.

write_context_info=

Signature: void **write_context_info=** (bool flag)

Description: Enables or disables context information

If this flag is set to false, no context information for PCell or library cell instances is written. Those cells will be converted to plain cells and KLayout will not be able to restore the identity of those cells. Use this option to enforce compatibility with other tools that don't understand the context information of KLayout.

The default value is true (context information is stored). Not all formats support context information, hence that flag has no effect for formats like CIF or DXF.

This method was introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'write_context_info'. This is the setter.

write_context_info?

Signature: *[const]* bool **write_context_info?**

Description: Gets a flag indicating whether context information will be stored

See write_context_info= for details about this flag.

This method was introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'write_context_info'. This is the getter.

4.54. API reference - Class LayoutQueryIterator

Notation used in Ruby API documentation

Module: db

Description: Provides the results of the query

This object is used by LayoutQuery#each to deliver the results of a query in an iterative fashion. See LayoutQuery for a detailed description of the query interface.

The LayoutQueryIterator class has been introduced in version 0.25.

Public constructors

new LayoutQueryIterator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	variant	**cell**		A shortcut for 'get("cell")'
	variant	**cell_index**		A shortcut for 'get("cell_index")'
	variant	**data**		A shortcut for 'get("data")'
	variant	**dtrans**		A shortcut for 'get("dtrans")'
	variant	**get**	(string name)	Gets the query property with the given name
	variant	**initial_cell**		A shortcut for 'get("initial_cell")'
	variant	**initial_cell_index**		A shortcut for 'get("initial_cell_index")'
	variant	**inst**		A shortcut for 'get("inst")'
	variant	**layer_index**		A shortcut for 'get("layer_index")'
[const]	const Layout ptr	**layout**		Gets the layout the query acts on
	variant	**parent_cell**		A shortcut for 'get("parent_cell")'
	variant	**parent_cell_index**		A shortcut for 'get("parent_cell_index")'

	variant	path_dtrans	A shortcut for 'get("path_dtrans")'
	variant	path_trans	A shortcut for 'get("path_trans")'
[const]	const LayoutQuery ptr	query	Gets the query the iterator follows on
	variant	shape	A shortcut for 'get("shape")'
	variant	trans	A shortcut for 'get("trans")'

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void _**manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the

	object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
cell	**Signature**: variant **cell** **Description**: A shortcut for 'get("cell")'
cell_index	**Signature**: variant **cell_index** **Description**: A shortcut for 'get("cell_index")'
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
data	**Signature**: variant **data** **Description**: A shortcut for 'get("data")'
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
dtrans	**Signature**: variant **dtrans** **Description**: A shortcut for 'get("dtrans")'

get	**Signature**: variant **get** (string name) **Description**: Gets the query property with the given name The query properties available can be obtained from the query object using LayoutQuery#property_names. Some shortcut methods are available. For example, the data method provides a shortcut for 'get("data")'. If a property with the given name is not available, nil will be returned.
initial_cell	**Signature**: variant **initial_cell** **Description**: A shortcut for 'get("initial_cell")'
initial_cell_index	**Signature**: variant **initial_cell_index** **Description**: A shortcut for 'get("initial_cell_index")'
inst	**Signature**: variant **inst** **Description**: A shortcut for 'get("inst")'
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
layer_index	**Signature**: variant **layer_index** **Description**: A shortcut for 'get("layer_index")'
layout	**Signature**: *[const]* const Layout ptr **layout** **Description**: Gets the layout the query acts on
new	**Signature**: *[static]* new LayoutQueryIterator ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
parent_cell	**Signature**: variant **parent_cell** **Description**: A shortcut for 'get("parent_cell")'
parent_cell_index	**Signature**: variant **parent_cell_index** **Description**: A shortcut for 'get("parent_cell_index")'
path_dtrans	**Signature**: variant **path_dtrans** **Description**: A shortcut for 'get("path_dtrans")'
path_trans	**Signature**: variant **path_trans** **Description**: A shortcut for 'get("path_trans")'

query	**Signature**: *[const]* const LayoutQuery ptr **query** **Description**: Gets the query the iterator follows on
shape	**Signature**: variant **shape** **Description**: A shortcut for 'get("shape")'
trans	**Signature**: variant **trans** **Description**: A shortcut for 'get("trans")'

4.55. API reference - Class LayoutQuery

Notation used in Ruby API documentation

Module: db

Description: A layout query

Layout queries are the backbone of the "Search & replace" feature. Layout queries allow retrieval of data from layouts and manipulation of layouts. This object provides script binding for this feature. Layout queries are used by first creating a query object. Depending on the nature of the query, either execute or each can be used to execute the query. execute will run the query and return once the query is finished. execute is useful for running queries that don't return results such as "delete" or "with ... do" queries. each can be used when the results of the query need to be retrieved.

The each method will call a block a of code for every result available. It will provide a LayoutQueryIterator object that allows accessing the results of the query. Depending on the query, different attributes of the iterator object will be available. For example, "select" queries will fill the "data" attribute with an array of values corresponding to the columns of the selection.

Here is some sample code:

```
ly = RBA::CellView::active.layout
q = RBA::LayoutQuery::new("select cell.name, cell.bbox from *")
q.each(ly) do |iter|
  puts "cell name: #{iter.data[0]}, bounding box: #{iter.data[1]}"
end
```

The LayoutQuery class has been introduced in version 0.25.

Public constructors

new LayoutQuery ptr	**new**	(string query)		Creates a new query object from the given query string

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const,iter]	LayoutQueryIterator	**each**	(const Layout ptr layout, ExpressionConte ptr context = nil)	Executes the query and delivered the results iteratively.
	void	**execute**	(Layout layout,	Executes the query

			ExpressionContext ptr context = nil)
[const]	string[]	**property_names**	Gets a list of property names available.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

each

Signature: *[const,iter]* LayoutQueryIterator **each** (const Layout ptr layout, ExpressionContext ptr context = nil)

Description: Executes the query and delivered the results iteratively.

The argument to the block is a LayoutQueryIterator object which can be asked for specific results.

The context argument allows supplying an expression execution context. This context can be used for example to supply variables for the execution. It has been added in version 0.26.

execute

Signature: void **execute** (Layout layout, ExpressionContext ptr context = nil)

Description: Executes the query

This method can be used to execute "active" queries such as "delete" or "with ... do". It is basically equivalent to iterating over the query until it is done.

The context argument allows supplying an expression execution context. This context can be used for example to supply variables for the execution. It has been added in version 0.26.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new LayoutQuery ptr **new** (string query)

Description: Creates a new query object from the given query string

Python specific notes:
This method is the default initializer of the object.

property_names

Signature: *[const]* string[] **property_names**

Description: Gets a list of property names available.

The list of properties available from the query depends on the nature of the query. This method allows detection of the properties available. Within the query, all of these properties can be obtained from the query iterator using LayoutQueryIterator#get.

4.56. API reference - Class Library

Notation used in Ruby API documentation

Module: db

Description: A Library

A library is basically a wrapper around a layout object. The layout object provides cells and potentially PCells that can be imported into other layouts.

The library provides a name which is used to identify the library and a description which is used for identifying the library in a user interface.

After a library is created and the layout is filled, it must be registered using the register method.

This class has been introduced in version 0.22.

Public constructors

new Library ptr	**new**		Creates a new, empty library

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add_technology**	(string tech)	Additionally associates the library with the given technology.
	void	**assign**	(const Library other)	Assigns another object to self
	void	**clear_technologies**		Clears the list of technologies the library is associated with.
	void	**delete**		Deletes the library
[const]	string	**description**		Returns the libraries' description text
	void	**description=**	(string description)	Sets the libraries' description text
[const]	new Library ptr	**dup**		Creates a copy of self
[const]	bool	**for_technologies**		Returns a value indicating whether the library is associated with any technology.

[const]	unsigned long	**id**		Returns the library's ID
[const]	bool	**is_for_technology**	(string tech)	Returns a value indicating whether the library is associated with the given technology.
	Layout	**layout**		The layout object where the cells reside that this library defines
[const]	Layout	**layout_const**		The layout object where the cells reside that this library defines (const version)
[const]	string	**name**		Returns the libraries' name
	void	**refresh**		Updates all layouts using this library.
	void	**register**	(string name)	Registers the library with the given name
[const]	string[]	**technologies**		Gets the list of technologies this library is associated with.
	void	**technology=**	(string technology)	sets the name of the technology the library is associated with

Public static methods and constants

Library ptr	**library_by_id**	(unsigned long id)	Gets the library object for the given ID
Library ptr	**library_by_name**	(string name, string for_technology = unspecific)	Gets a library by name
unsigned long[]	**library_ids**		Returns a list of valid library IDs.
string[]	**library_names**		Returns a list of the names of all libraries registered in the system.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead
	string	**technology**	Use of this method is deprecated

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

	Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
add_technology	**Signature**: void **add_technology** (string tech) **Description**: Additionally associates the library with the given technology. See also clear_technologies. This method has been introduced in version 0.27
assign	**Signature**: void **assign** (const Library other) **Description**: Assigns another object to self
clear_technologies	**Signature**: void **clear_technologies** **Description**: Clears the list of technologies the library is associated with.

See also add_technology.

This method has been introduced in version 0.27

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

delete

Signature: void **delete**

Description: Deletes the library

This method will delete the library object. Library proxies pointing to this library will become invalid and the library object cannot be used any more after calling this method.

This method has been introduced in version 0.25.

description

Signature: *[const]* string **description**

Description: Returns the libraries' description text

Python specific notes:
The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string description)

Description: Sets the libraries' description text

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Library ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

for_technologies

Signature: *[const]* bool **for_technologies**

Description: Returns a value indicating whether the library is associated with any technology.

	The method is equivalent to checking whether the technologies list is empty. This method has been introduced in version 0.27
id	**Signature**: *[const]* unsigned long **id** **Description**: Returns the library's ID The ID is set when the library is registered and cannot be changed
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
is_for_technology	**Signature**: *[const]* bool **is_for_technology** (string tech) **Description**: Returns a value indicating whether the library is associated with the given technology. This method has been introduced in version 0.27
layout	**Signature**: Layout **layout** **Description**: The layout object where the cells reside that this library defines
layout_const	**Signature**: *[const]* Layout **layout_const** **Description**: The layout object where the cells reside that this library defines (const version)
library_by_id	**Signature**: *[static]* Library ptr **library_by_id** (unsigned long id) **Description**: Gets the library object for the given ID If the ID is not valid, nil is returned. This method has been introduced in version 0.27.
library_by_name	**Signature**: *[static]* Library ptr **library_by_name** (string name, string for_technology = unspecific) **Description**: Gets a library by name Returns the library object for the given name. If the name is not a valid library name, nil is returned. Different libraries can be registered under the same names for different technologies. When a technology name is given in 'for_technologies', the first library matching this technology is returned. If no technology is given, the first library is returned. The technology selector has been introduced in version 0.27.
library_ids	**Signature**: *[static]* unsigned long[] **library_ids** **Description**: Returns a list of valid library IDs. See library_names for the reasoning behind this method. This method has been introduced in version 0.27.
library_names	**Signature**: *[static]* string[] **library_names** **Description**: Returns a list of the names of all libraries registered in the system. NOTE: starting with version 0.27, the name of a library does not need to be unique if libraries are associated with specific technologies. This method will only return the names and it's not possible

not unambiguously derive the library object. It is recommended to use library_ids and library_by_id to obtain the library unambiguously.

name

Signature: *[const]* string **name**

Description: Returns the libraries' name

The name is set when the library is registered and cannot be changed

new

Signature: *[static]* new Library ptr **new**

Description: Creates a new, empty library

Python specific notes:
This method is the default initializer of the object.

refresh

Signature: void **refresh**

Description: Updates all layouts using this library.

This method will retire cells or update layouts in the attached clients. It will also recompute the PCells inside the library. This method has been introduced in version 0.27.8.

register

Signature: void **register** (string name)

Description: Registers the library with the given name

This method can be called in the constructor to register the library after the layout object has been filled with content. If a library with that name already exists for the same technologies, it will be replaced with this library.

This method will set the libraries' name.

The technology specific behaviour has been introduced in version 0.27.

technologies

Signature: *[const]* string[] **technologies**

Description: Gets the list of technologies this library is associated with.

This method has been introduced in version 0.27

technology

Signature: string **technology**

Description: Returns name of the technology the library is associated with

Use of this method is deprecated

If this attribute is a non-empty string, this library is only offered for selection if the current layout uses this technology.

This attribute has been introduced in version 0.25. In version 0.27 this attribute is deprecated as a library can now be associated with multiple technologies.

Python specific notes:
The object exposes a readable attribute 'technology'. This is the getter.

technology=

Signature: void **technology=** (string technology)

Description: sets the name of the technology the library is associated with

See technology for details. This attribute has been introduced in version 0.25. In version 0.27, a library can be associated with multiple technologies and this method will revert the selection to a single one. Passing an empty string is equivalent to clear_technologies.

Python specific notes:
The object exposes a writable attribute 'technology'. This is the setter.

4.57. API reference - Class PCellParameterState

Notation used in Ruby API documentation

Module: db

Description: Provides access to the attributes of a single parameter within \PCellParameterStates.

Sub-classes: ParameterStateIcon

See PCellParameterStates for details about this feature.

This class has been introduced in version 0.28.

Public constructors

new PCellParameterState ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const PCellParameterState other)	Assigns another object to self
[const]	new PCellParameterState ptr	**dup**		Creates a copy of self
	void	**enabled=**	(bool f)	Sets a value indicating whether the parameter is enabled in the parameter form
	void	**icon=**	(PCellParameterStateIcon i)	Sets the icon for the parameter
[const]	bool	**is_enabled?**		Gets a value indicating whether the parameter is enabled in the parameter form
[const]	bool	**is_readonly?**		Gets a value indicating whether the parameter is read-only (not editable) in the parameter form
[const]	bool	**is_visible?**		Gets a value indicating whether the parameter is visible in the parameter form

	void	**readonly=**	(bool f)	Sets a value indicating whether the parameter is made read-only (not editable) in the parameter form
[const]	string	**tooltip**		Gets the tool tip text
[const]	string	**tooltip**		Gets the icon for the parameter
	void	**tooltip=**	(string s)	Sets the tool tip text
[const]	variant	**value**		Gets the value of the parameter
	void	**value=**	(variant v)	Sets the value of the parameter
	void	**visible=**	(bool f)	Sets a value indicating whether the parameter is visible in the parameter form

Public static methods and constants

[static,const]	PCellParameterState::ParameterStateIc	**ErrorIcon**	An icon indicating an error is shown
[static,const]	PCellParameterState::ParameterStateIcon	**InfoIcon**	A general 'information' icon is shown
[static,const]	PCellParameterState::ParameterStateIc	**NoIcon**	No icon is shown for the parameter
[static,const]	PCellParameterState::ParameterStateIcon	**WarningIcon**	An icon indicating a warning is shown

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

ErrorIcon

Signature: *[static,const]* PCellParameterState::ParameterStateIcon **ErrorIcon**

Description: An icon indicating an error is shown

Python specific notes:
The object exposes a readable attribute 'ErrorIcon'. This is the getter.

InfoIcon

Signature: *[static,const]* PCellParameterState::ParameterStateIcon **InfoIcon**

Description: A general 'information' icon is shown

Python specific notes:
The object exposes a readable attribute 'InfoIcon'. This is the getter.

NoIcon

Signature: *[static,const]* PCellParameterState::ParameterStateIcon **NoIcon**

Description: No icon is shown for the parameter

Python specific notes:
The object exposes a readable attribute 'NoIcon'. This is the getter.

WarningIcon

Signature: *[static,const]* PCellParameterState::ParameterStateIcon **WarningIcon**

Description: An icon indicating a warning is shown

Python specific notes:
The object exposes a readable attribute 'WarningIcon'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const PCellParameterState other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new PCellParameterState ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

enabled=

Signature: void **enabled=** (bool f)

Description: Sets a value indicating whether the parameter is enabled in the parameter form

Python specific notes:
The object exposes a writable attribute 'enabled'. This is the setter.

icon=

Signature: void **icon=** (PCellParameterState::ParameterStateIcon i)

Description: Sets the icon for the parameter

Python specific notes:
The object exposes a writable attribute 'icon'. This is the setter.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_enabled?

Signature: *[const]* bool **is_enabled?**

Description: Gets a value indicating whether the parameter is enabled in the parameter form

Python specific notes:
The object exposes a readable attribute 'enabled'. This is the getter.

is_readonly?

Signature: *[const]* bool **is_readonly?**

Description: Gets a value indicating whether the parameter is read-only (not editable) in the parameter form

Python specific notes:
The object exposes a readable attribute 'readonly'. This is the getter.

is_visible?

Signature: *[const]* bool **is_visible?**

Description: Gets a value indicating whether the parameter is visible in the parameter form

Python specific notes:
The object exposes a readable attribute 'visible'. This is the getter.

new

Signature: *[static]* new PCellParameterState ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

readonly=

Signature: void **readonly=** (bool f)

Description: Sets a value indicating whether the parameter is made read-only (not editable) in the parameter form

Python specific notes:
The object exposes a writable attribute 'readonly'. This is the setter.

tooltip

(1) Signature: *[const]* string **tooltip**

Description: Gets the tool tip text

Python specific notes:
The object exposes a readable attribute 'tooltip'. This is the getter.

(2) Signature: *[const]* string **tooltip**

Description: Gets the icon for the parameter

Python specific notes:
This method is available as 'tooltip_' in Python to distiguish it from the property with the same name.

tooltip=

Signature: void **tooltip=** (string s)

Description: Sets the tool tip text

The tool tip is shown when hovering over the parameter label or edit field.

Python specific notes:
The object exposes a writable attribute 'tooltip'. This is the setter.

value

Signature: *[const]* variant **value**

Description: Gets the value of the parameter

Python specific notes:

	The object exposes a readable attribute 'value'. This is the getter.
value=	**Signature**: void **value=** (variant v) **Description**: Sets the value of the parameter **Python specific notes:** The object exposes a writable attribute 'value'. This is the setter.
visible=	**Signature**: void **visible=** (bool f) **Description**: Sets a value indicating whether the parameter is visible in the parameter form **Python specific notes:** The object exposes a writable attribute 'visible'. This is the setter.

4.58. API reference - Class PCellParameterState::ParameterStateIcon

Notation used in Ruby API documentation

Module: db

Description: This enum specifies the icon shown next to the parameter in PCell parameter list.

This class is equivalent to the class PCellParameterState::ParameterStateIcon

This enum was introduced in version 0.28.

Public constructors

new PCellParameterState::ParameterStateIcon ptr	**new**	(int i)	Creates an enum from an integer value
new PCellParameterState::ParameterStateIcon ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const PCellParameterState::Para other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const PCellParameterState::Para other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const PCellParameterState::Para other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	PCellParameterState::ParameterStateIc	**ErrorIcon**	An icon indicating an error is shown

[static,const]	PCellParameterState::ParameterStateIcon	**InfoIcon**	A general 'information' icon is shown
[static,const]	PCellParameterState::ParameterStateIc	**NoIcon**	No icon is shown for the parameter
[static,const]	PCellParameterState::ParameterStateIcon	**WarningIcon**	An icon indicating a warning is shown

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const PCellParameterState::ParameterStateIcon other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const PCellParameterState::ParameterStateIcon other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const PCellParameterState::ParameterStateIcon other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
ErrorIcon	**Signature**: *[static,const]* PCellParameterState::ParameterStateIcon **ErrorIcon** **Description**: An icon indicating an error is shown **Python specific notes:** The object exposes a readable attribute 'ErrorIcon'. This is the getter.
InfoIcon	**Signature**: *[static,const]* PCellParameterState::ParameterStateIcon **InfoIcon** **Description**: A general 'information' icon is shown **Python specific notes:** The object exposes a readable attribute 'InfoIcon'. This is the getter.
NoIcon	**Signature**: *[static,const]* PCellParameterState::ParameterStateIcon **NoIcon** **Description**: No icon is shown for the parameter **Python specific notes:** The object exposes a readable attribute 'NoIcon'. This is the getter.
WarningIcon	**Signature**: *[static,const]* PCellParameterState::ParameterStateIcon **WarningIcon** **Description**: An icon indicating a warning is shown **Python specific notes:**

	The object exposes a readable attribute 'WarningIcon'. This is the getter.
hash	**Signature**: *[const]* int **hash** **Description**: Gets the hash value from the enum **Python specific notes:** This method is also available as 'hash(object)'.
inspect	**Signature**: *[const]* string **inspect** **Description**: Converts an enum to a visual string **Python specific notes:** This method is also available as 'repr(object)'.
new	**(1) Signature**: *[static]* new PCellParameterState::ParameterStateIcon ptr **new** (int i) **Description**: Creates an enum from an integer value **Python specific notes:** This method is the default initializer of the object. **(2) Signature**: *[static]* new PCellParameterState::ParameterStateIcon ptr **new** (string s) **Description**: Creates an enum from a string value **Python specific notes:** This method is the default initializer of the object.
to_i	**Signature**: *[const]* int **to_i** **Description**: Gets the integer value from the enum **Python specific notes:** This method is also available as 'int(object)'.
to_s	**Signature**: *[const]* string **to_s** **Description**: Gets the symbolic string from an enum **Python specific notes:** This method is also available as 'str(object)'.

4.59. API reference - Class PCellParameterStates

Notation used in Ruby API documentation

Module: db

Description: Provides access to the parameter states inside a 'callback' implementation of a PCell

Example: enables or disables a parameter 'n' based on the value:

```
n_param = states.parameter("n")
n_param.enabled = n_param.value > 1.0
```

This class has been introduced in version 0.28.

Public constructors

new PCellParameterStates ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const PCellParamet other)	Assigns another object to self
[const]	new PCellParameterStates ptr	**dup**		Creates a copy of self
[const]	bool	**has_parameter?**	(string name)	Gets a value indicating whether a parameter with that name exists
	PCellParameterState	**parameter**	(string name)	Gets the parameter by name

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead

[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const PCellParameterStates other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new PCellParameterStates ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

has_parameter?

Signature: *[const]* bool **has_parameter?** (string name)

Description: Gets a value indicating whether a parameter with that name exists

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new PCellParameterStates ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

parameter

Signature: PCellParameterState **parameter** (string name)

Description: Gets the parameter by name

This will return a PCellParameterState object that can be used to manipulate the parameter state.

4.60. API reference - Class PCellDeclaration

Notation used in Ruby API documentation

Module: db

Description: A PCell declaration providing the parameters and code to produce the PCell

Class hierarchy: PCellDeclaration

A PCell declaration is basically the recipe of how to create a PCell layout from a parameter set. The declaration includes

- Parameters: names, types, default values
- Layers: the layers the PCell wants to create
- Code: a production callback that is called whenever a PCell is instantiated with a certain parameter set
- Display name: the name that is shown for a given PCell instance

All these declarations are implemented by deriving from the PCellDeclaration class and reimplementing the specific methods. Reimplementing the display_name method is optional. The default implementation creates a name from the PCell name plus the parameters.

By supplying the information about the layers it wants to create, KLayout is able to call the production callback with a defined set of the layer ID's which are already mapped to valid actual layout layers.

This class has been introduced in version 0.22.

Public constructors

new PCellDeclaration ptr	**new**		Creates a new object of this class

Public methods

	void	**_assign**	(const PCellDeclaration other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new PCellDeclaration ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

	void	assign	(const PCellDeclaration other)	Assigns another object to self
[virtual,const]	void	callback	(const Layout layout, string name, PCellParameterStates states)	Indicates a parameter change and allows implementing actions based on the parameter value
[virtual,const]	bool	can_create_from_sha	(const Layout layout, const Shape shape, unsigned int layer)	Returns true, if the PCell can be created from the given shape
[virtual,const]	variant[]	coerce_parameters	(const Layout layout, variant[] input)	Modifies the parameters to match the requirements
[virtual,const]	string	display_text	(variant[] parameters)	Returns the display text for this PCell given a certain parameter set
[const]	new PCellDeclaration ptr	dup		Creates a copy of self
[virtual,const]	LayerInfo[]	get_layers	(variant[] parameters)	Returns a list of layer declarations
[virtual,const]	PCellParameterDeclaration[]	get_parameters		Returns a list of parameter declarations
[const]	unsigned int	id		Gets the integer ID of the PCell declaration
[const]	Layout ptr	layout		Gets the Layout object the PCell is registered in or nil if it is not registered yet.
[const]	string	name		Gets the name of the PCell
[virtual,const]	variant[]	parameters_from_shape	(const Layout layout, const Shape shape, unsigned int layer)	Gets the parameters for the PCell which can replace the given shape
[virtual,const]	void	produce	(const Layout layout, unsigned int[] layer_ids, variant[] parameters, Cell cell)	The production callback
[virtual,const]	Trans	transformation_from_shape	(const Layout layout, const Shape shape,	Gets the instance transformation for the PCell which can replace the given shape

unsigned int layer)

[virtual,const]	bool	**wants_lazy_evaluatic**	Gets a value indicating whether the PCell wants lazy evaluation

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_assign

Signature: void **_assign** (const PCellDeclaration other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new PCellDeclaration ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const PCellDeclaration other)

Description: Assigns another object to self

callback

Signature: *[virtual,const]* void **callback** (const Layout layout, string name, PCellParameterStates states)

Description: Indicates a parameter change and allows implementing actions based on the parameter value

layout:	The layout object in which the PCell will be produced
name:	The name of the parameter which has changed or an empty string if all parameters need to be considered
states:	A PCellParameterStates object which can be used to manipulate the parameter states

This method may be reimplemented to implement parameter-specific actions upon value change or button callbacks. Whenever the value of a parameter is changed in the PCell parameter form, this method is called with the name of the parameter in 'name'. The implementation can manipulate values or states (enabled, visible) or parameters using the PCellParameterStates object passed in 'states'.

Initially, this method will be called with an empty parameter name to indicate a global change. The implementation may then consolidate all states. The initial state is build from the 'readonly' (disabled) or 'hidden' (invisible) parameter declarations.

This method is also called when a button-type parameter is present and the button is pressed. In this case the parameter name is the name of the button.

This feature has been introduced in version 0.28.

can_create_from_shape

Signature: *[virtual,const]* bool **can_create_from_shape** (const Layout layout, const Shape shape, unsigned int layer)

Description: Returns true, if the PCell can be created from the given shape

layout:	The layout the shape lives in
shape:	The shape from which a PCell shall be created
layer:	The layer index (in layout) of the shape

KLayout offers a way to convert a shape into a PCell. To test whether the PCell can be created from a shape, it will call this method. If this method returns true, KLayout will use parameters_from_shape

and transformation_from_shape to derive the parameters and instance transformation for the new PCell instance that will replace the shape.

coerce_parameters

Signature: *[virtual,const]* variant[] **coerce_parameters** (const Layout layout, variant[] input)

Description: Modifies the parameters to match the requirements

layout:	The layout object in which the PCell will be produced
input:	The parameters before the modification
Returns:	The modified parameters or an empty array, indicating that no modification was done

This method can be reimplemented to change the parameter set according to some constraints for example. The reimplementation may modify the parameters in a way that they are usable for the produce method.

The method receives a reference to the layout so it is able to verify the parameters against layout properties.

It can raise an exception to indicate that something is not correct.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

display_text

Signature: *[virtual,const]* string **display_text** (variant[] parameters)

Description: Returns the display text for this PCell given a certain parameter set

Reimplement this method to create a distinct display text for a PCell variant with the given parameter set. If this method is not implemented, a default text is created.

dup

Signature: *[const]* new PCellDeclaration ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

get_layers

Signature: *[virtual,const]* LayerInfo[] **get_layers** (variant[] parameters)

Description: Returns a list of layer declarations

Reimplement this method to return a list of layers this PCell wants to create. The layer declarations are returned as a list of LayerInfo objects which are used as match expressions to look up the layer in the actual layout.

This method receives the PCell parameters which allows it to deduce layers from the parameters.

get_parameters

Signature: *[virtual,const]* PCellParameterDeclaration[] **get_parameters**

Description: Returns a list of parameter declarations

Reimplement this method to return a list of parameters used in that PCell implementation. A parameter declaration is a PCellParameterDeclaration object and defines the parameter name, type, description text and possible choices for the parameter value.

id

Signature: *[const]* unsigned int **id**

Description: Gets the integer ID of the PCell declaration

This ID is used to identify the PCell in the context of a Layout object for example

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

layout

Signature: *[const]* Layout ptr **layout**

Description: Gets the Layout object the PCell is registered in or nil if it is not registered yet.

This attribute has been added in version 0.27.5.

name

Signature: *[const]* string **name**

Description: Gets the name of the PCell

new

Signature: *[static]* new PCellDeclaration ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

parameters_from_shape

Signature: *[virtual,const]* variant[] **parameters_from_shape** (const Layout layout, const Shape shape, unsigned int layer)

Description: Gets the parameters for the PCell which can replace the given shape

layout:	The layout the shape lives in
shape:	The shape from which a PCell shall be created
layer:	The layer index (in layout) of the shape

KLayout offers a way to convert a shape into a PCell. If can_create_from_shape returns true, it will use this method to derive the parameters for the PCell instance that will replace the shape. See also transformation_from_shape and can_create_from_shape.

produce

Signature: *[virtual,const]* void **produce** (const Layout layout, unsigned int[] layer_ids, variant[] parameters, Cell cell)

Description: The production callback

layout:	The layout object where the cell resides
layer_ids:	A list of layer ID's which correspond to the layers declared with get_layers
parameters:	A list of parameter values which correspond to the parameters declared with get_parameters
cell:	The cell where the layout will be created

Reimplement this method to provide the code that implements the PCell. The code is supposed to create the layout in the target cell using the provided parameters and the layers passed in the layer_ids list.

transformation_from_shap

Signature: *[virtual,const]* Trans **transformation_from_shape** (const Layout layout, const Shape shape, unsigned int layer)

Description: Gets the instance transformation for the PCell which can replace the given shape

layout:	The layout the shape lives in
shape:	The shape from which a PCell shall be created
layer:	The layer index (in layout) of the shape

KLayout offers a way to convert a shape into a PCell. If can_create_from_shape returns true, it will use this method to derive the transformation for the PCell instance that will replace the shape. See also parameters_from_shape and can_create_from_shape.

wants_lazy_evaluation

Signature: *[virtual,const]* bool **wants_lazy_evaluation**

Description: Gets a value indicating whether the PCell wants lazy evaluation

In lazy evaluation mode, the PCell UI will not immediately update the layout when a parameter is changed. Instead, the user has to commit the changes in order to have the parameters updated. This is useful for PCells that take a long time to compute.

The default implementation will return 'false' indicating immediate updates.

This method has been added in version 0.27.6.

4.61. API reference - Class PCellParameterDeclaration

Notation used in Ruby API documentation

Module: db

Description: A PCell parameter declaration

This class declares a PCell parameter by providing a name, the type and a value and additional information like description, unit string and default value. It is used in the PCellDeclaration class to deliver the necessary information.

This class has been introduced in version 0.22.

Public constructors

new PCellParameterDeclaration ptr	**new**	(string name, unsigned int type, string description, variant default = nil, string unit =)	Create a new parameter declaration with the given name, type, default value and unit string

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add_choice**	(string description, variant value)	Add a new value to the list of choices
	void	**assign**	(const PCellParameterDeclaration other)	Assigns another object to self
[const]	string[]	**choice_descriptions**		Returns a list of choice descriptions
[const]	variant[]	**choice_values**		Returns a list of choice values
	void	**clear_choices**		Clears the list of choices
[const]	variant	**default**		Gets the default value
	void	**default=**	(variant value)	Sets the default value
[const]	string	**description**		Gets the description text

	void	**description=**	(string description)	Sets the description
[const]	new PCellParameterDeclaration ptr	**dup**		Creates a copy of self
	void	**hidden=**	(bool flag)	Makes the parameter hidden if this attribute is set to true
[const]	bool	**hidden?**		Returns true, if the parameter is a hidden parameter that should not be shown in the user interface
[const]	variant	**max_value**		Gets the maximum value allowed
	void	**max_value=**	(variant value)	Sets the maximum value allowed
[const]	variant	**min_value**		Gets the minimum value allowed
	void	**min_value=**	(variant value)	Sets the minimum value allowed
[const]	string	**name**		Gets the name
	void	**name=**	(string value)	Sets the name
	void	**readonly=**	(bool flag)	Makes the parameter read-only if this attribute is set to true
[const]	bool	**readonly?**		Returns true, if the parameter is a read-only parameter
[const]	unsigned int	**type**		Gets the type
	void	**type=**	(unsigned int type)	Sets the type
[const]	string	**unit**		Gets the unit string
	void	**unit=**	(string unit)	Sets the unit string

Public static methods and constants

unsigned int	**TypeBoolean**	Type code: boolean data
unsigned int	**TypeCallback**	Type code: a button triggering a callback
unsigned int	**TypeDouble**	Type code: floating-point data
unsigned int	**TypeInt**	Type code: integer data
unsigned int	**TypeLayer**	Type code: a layer (a LayerInfo object)
unsigned int	**TypeList**	Type code: a list of variants
unsigned int	**TypeNone**	Type code: unspecific type

unsigned int	**TypeShape**	Type code: a guiding shape (Box, Edge, Point, Polygon or Path)
unsigned int	**TypeString**	Type code: string data

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

TypeBoolean

Signature: *[static]* unsigned int **TypeBoolean**

Description: Type code: boolean data

Python specific notes:
The object exposes a readable attribute 'TypeBoolean'. This is the getter.

TypeCallback

Signature: *[static]* unsigned int **TypeCallback**

Description: Type code: a button triggering a callback

This code has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'TypeCallback'. This is the getter.

TypeDouble

Signature: *[static]* unsigned int **TypeDouble**

Description: Type code: floating-point data

Python specific notes:
The object exposes a readable attribute 'TypeDouble'. This is the getter.

TypeInt

Signature: *[static]* unsigned int **TypeInt**

Description: Type code: integer data

Python specific notes:
The object exposes a readable attribute 'TypeInt'. This is the getter.

TypeLayer

Signature: *[static]* unsigned int **TypeLayer**

Description: Type code: a layer (a LayerInfo object)

Python specific notes:
The object exposes a readable attribute 'TypeLayer'. This is the getter.

TypeList

Signature: *[static]* unsigned int **TypeList**

Description: Type code: a list of variants

Python specific notes:

The object exposes a readable attribute 'TypeList'. This is the getter.

TypeNone

Signature: *[static]* unsigned int **TypeNone**

Description: Type code: unspecific type

Python specific notes:
The object exposes a readable attribute 'TypeNone'. This is the getter.

TypeShape

Signature: *[static]* unsigned int **TypeShape**

Description: Type code: a guiding shape (Box, Edge, Point, Polygon or Path)

Python specific notes:
The object exposes a readable attribute 'TypeShape'. This is the getter.

TypeString

Signature: *[static]* unsigned int **TypeString**

Description: Type code: string data

Python specific notes:
The object exposes a readable attribute 'TypeString'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
add_choice	**Signature**: void **add_choice** (string description, variant value) **Description**: Add a new value to the list of choices This method will add the given value with the given description to the list of choices. If choices are defined, KLayout will show a drop-down box instead of an entry field in the parameter user interface.
assign	**Signature**: void **assign** (const PCellParameterDeclaration other) **Description**: Assigns another object to self
choice_descriptions	**Signature**: *[const]* string[] **choice_descriptions** **Description**: Returns a list of choice descriptions
choice_values	**Signature**: *[const]* variant[] **choice_values** **Description**: Returns a list of choice values
clear_choices	**Signature**: void **clear_choices** **Description**: Clears the list of choices
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
default	**Signature**: *[const]* variant **default** **Description**: Gets the default value **Python specific notes:** The object exposes a readable attribute 'default'. This is the getter.
default=	**Signature**: void **default=** (variant value) **Description**: Sets the default value If a default value is defined, it will be used to initialize the parameter value when a PCell is created. **Python specific notes:** The object exposes a writable attribute 'default'. This is the setter.

description

Signature: *[const]* string **description**

Description: Gets the description text

Python specific notes:
The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string description)

Description: Sets the description

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new PCellParameterDeclaration ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

hidden=

Signature: void **hidden=** (bool flag)

Description: Makes the parameter hidden if this attribute is set to true

Python specific notes:
The object exposes a writable attribute 'hidden'. This is the setter.

hidden?

Signature: *[const]* bool **hidden?**

Description: Returns true, if the parameter is a hidden parameter that should not be shown in the user interface

By making a parameter hidden, it is possible to create internal parameters which cannot be edited.

Python specific notes:
The object exposes a readable attribute 'hidden'. This is the getter.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

max_value

Signature: *[const]* variant **max_value**

Description: Gets the maximum value allowed

See max_value= for a description of this attribute.

This attribute has been added in version 0.29.

Python specific notes:
The object exposes a readable attribute 'max_value'. This is the getter.

max_value=

Signature: void **max_value=** (variant value)

Description: Sets the maximum value allowed

The maximum value is a visual feature and limits the allowed values for numerical entry boxes. This applies to parameters of type int or double. The maximum value is not effective if choices are present.

The maximum value is not enforced - for example there is no restriction implemented when setting values programmatically.

Setting this attribute to "nil" (the default) implies "no limit".

This attribute has been added in version 0.29.

Python specific notes:
The object exposes a writable attribute 'max_value'. This is the setter.

min_value

Signature: *[const]* variant **min_value**

Description: Gets the minimum value allowed

See min_value= for a description of this attribute.

This attribute has been added in version 0.29.

Python specific notes:
The object exposes a readable attribute 'min_value'. This is the getter.

min_value=

Signature: void **min_value=** (variant value)

Description: Sets the minimum value allowed

The minimum value is a visual feature and limits the allowed values for numerical entry boxes. This applies to parameters of type int or double. The minimum value is not effective if choices are present.

The minimum value is not enforced - for example there is no restriction implemented when setting values programmatically.

Setting this attribute to "nil" (the default) implies "no limit".

This attribute has been added in version 0.29.

Python specific notes:
The object exposes a writable attribute 'min_value'. This is the setter.

name

Signature: *[const]* string **name**

Description: Gets the name

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string value)

Description: Sets the name

Python specific notes:

The object exposes a writable attribute 'name'. This is the setter.

new

Signature: *[static]* new PCellParameterDeclaration ptr **new** (string name, unsigned int type, string description, variant default = nil, string unit =)

Description: Create a new parameter declaration with the given name, type, default value and unit string

name:	The parameter name
type:	One of the Type... constants describing the type of the parameter
description:	The description text
default:	The default (initial) value
unit:	The unit string

Python specific notes:
This method is the default initializer of the object.

readonly=

Signature: void **readonly=** (bool flag)

Description: Makes the parameter read-only if this attribute is set to true

Python specific notes:
The object exposes a writable attribute 'readonly'. This is the setter.

readonly?

Signature: *[const]* bool **readonly?**

Description: Returns true, if the parameter is a read-only parameter

By making a parameter read-only, it is shown but cannot be edited.

Python specific notes:
The object exposes a readable attribute 'readonly'. This is the getter.

type

Signature: *[const]* unsigned int **type**

Description: Gets the type

The type is one of the T... constants.

Python specific notes:
The object exposes a readable attribute 'type'. This is the getter.

type=

Signature: void **type=** (unsigned int type)

Description: Sets the type

Python specific notes:
The object exposes a writable attribute 'type'. This is the setter.

unit

Signature: *[const]* string **unit**

Description: Gets the unit string

Python specific notes:
The object exposes a readable attribute 'unit'. This is the getter.

unit=

Signature: void **unit=** (string unit)

Description: Sets the unit string

The unit string is shown right to the edit fields for numeric parameters.

Python specific notes:

The object exposes a writable attribute 'unit'. This is the setter.

4.62. API reference - Class LogEntryData

Notation used in Ruby API documentation

Module: db

Description: A generic log entry

This class is used for example by the device extractor (see NetlistDeviceExtractor) to keep errors or warnings that occurred during extraction of the devices.

Other classes also make use of this object to store errors, warnings or information. The log entry object features a severity (warning, error, info), a message, an optional category name and description (good for filtering if needed) and an optional DPolygon object for indicating some location or error marker. The original class used to be "NetlistDeviceExtractorError" which had been introduced in version 0.26. It was generalized and renamed in version 0.28.13 as it was basically not useful as a separate class.

Public constructors

new LogEntryData ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const LogEntryData other)	Assigns another object to self
[const]	string	**category_description**		Gets the category description.
	void	**category_description=**	(string description)	Sets the category description.
[const]	string	**category_name**		Gets the category name.
	void	**category_name=**	(string name)	Sets the category name.
[const]	string	**cell_name**		Gets the cell name.
	void	**cell_name=**	(string cell_name)	Sets the cell name.
[const]	new LogEntryData ptr	**dup**		Creates a copy of self

[const]	DPolygon	**geometry**		Gets the geometry.
	void	**geometry=**	(const DPolygon polygon)	Sets the geometry.
[const]	string	**message**		Gets the message text.
	void	**message=**	(string message)	Sets the message text.
[const]	Severity	**severity**		Gets the severity attribute.
	void	**severity=**	(Severity severity)	Sets the severity attribute.
[const]	string	**to_s**	(bool with_geometry = true)	Gets the string representation of this error or warning.

Public static methods and constants

[static,const]	Severity	**Error**	Specifies error severity (preferred action is stop)
[static,const]	Severity	**Info**	Specifies info severity (print if requested, otherwise silent)
[static,const]	Severity	**NoSeverity**	Specifies no particular severity (default)
[static,const]	Severity	**Warning**	Specifies warning severity (log with high priority, but do not stop)

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

Error

Signature: *[static,const]* Severity **Error**

Description: Specifies error severity (preferred action is stop)

Python specific notes:
The object exposes a readable attribute 'Error'. This is the getter.

Info

Signature: *[static,const]* Severity **Info**

Description: Specifies info severity (print if requested, otherwise silent)

Python specific notes:

The object exposes a readable attribute 'Info'. This is the getter.

NoSeverity

Signature: *[static,const]* Severity **NoSeverity**

Description: Specifies no particular severity (default)

Python specific notes:
The object exposes a readable attribute 'NoSeverity'. This is the getter.

Warning

Signature: *[static,const]* Severity **Warning**

Description: Specifies warning severity (log with high priority, but do not stop)

Python specific notes:
The object exposes a readable attribute 'Warning'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the

	reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
assign	**Signature**: void **assign** (const LogEntryData other) **Description**: Assigns another object to self
category_description	**Signature**: *[const]* string **category_description** **Description**: Gets the category description. See category_name= for details about categories. **Python specific notes:** The object exposes a readable attribute 'category_description'. This is the getter.
category_description=	**Signature**: void **category_description=** (string description) **Description**: Sets the category description. See category_name= for details about categories. **Python specific notes:** The object exposes a writable attribute 'category_description'. This is the setter.
category_name	**Signature**: *[const]* string **category_name** **Description**: Gets the category name. See category_name= for more details. **Python specific notes:** The object exposes a readable attribute 'category_name'. This is the getter.
category_name=	**Signature**: void **category_name=** (string name) **Description**: Sets the category name. The category name is optional. If given, it specifies a formal category name. Errors with the same category name are shown in that category. If in addition a category description is specified (see category_description), this description will be displayed as the title. **Python specific notes:** The object exposes a writable attribute 'category_name'. This is the setter.
cell_name	**Signature**: *[const]* string **cell_name** **Description**: Gets the cell name. See cell_name= for details about this attribute. **Python specific notes:** The object exposes a readable attribute 'cell_name'. This is the getter.
cell_name=	**Signature**: void **cell_name=** (string cell_name) **Description**: Sets the cell name. The cell (or circuit) name specifies the cell or circuit the log entry is related to. If the log entry is an error or warning generated during device extraction, the cell name is the circuit the device should have appeared in. **Python specific notes:** The object exposes a writable attribute 'cell_name'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new LogEntryData ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

geometry

Signature: *[const]* DPolygon **geometry**

Description: Gets the geometry.

See geometry= for more details.

Python specific notes:
The object exposes a readable attribute 'geometry'. This is the getter.

geometry=

Signature: void **geometry=** (const DPolygon polygon)

Description: Sets the geometry.

The geometry is optional. If given, a marker may be shown when selecting this error.

Python specific notes:
The object exposes a writable attribute 'geometry'. This is the setter.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

message

Signature: *[const]* string **message**

Description: Gets the message text.

Python specific notes:

	The object exposes a readable attribute 'message'. This is the getter.
message=	**Signature**: void **message=** (string message) **Description**: Sets the message text. **Python specific notes:** The object exposes a writable attribute 'message'. This is the setter.
new	**Signature**: *[static]* new LogEntryData ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
severity	**Signature**: *[const]* Severity **severity** **Description**: Gets the severity attribute. **Python specific notes:** The object exposes a readable attribute 'severity'. This is the getter.
severity=	**Signature**: void **severity=** (Severity severity) **Description**: Sets the severity attribute. **Python specific notes:** The object exposes a writable attribute 'severity'. This is the setter.
to_s	**Signature**: *[const]* string **to_s** (bool with_geometry = true) **Description**: Gets the string representation of this error or warning. This method has been introduced in version 0.28.13. **Python specific notes:** This method is also available as 'str(object)'.

4.63. API reference - Class Severity

Notation used in Ruby API documentation

Module: db

Description: This enum specifies the severity level for log entries.

This enum was introduced in version 0.28.13.

Public constructors

new Severity ptr	**new**	(int i)	Creates an enum from an integer value
new Severity ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const Severity other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const Severity other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const Severity other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Severity other)	Assigns another object to self
[const]	new Severity ptr	**dup**		Creates a copy of self

[const]	int	**hash**	Gets the hash value from the enum
[const]	string	**inspect**	Converts an enum to a visual string
[const]	int	**to_i**	Gets the integer value from the enum
[const]	string	**to_s**	Gets the symbolic string from an enum

Public static methods and constants

[static,const]	Severity	**Error**	Specifies error severity (preferred action is stop)
[static,const]	Severity	**Info**	Specifies info severity (print if requested, otherwise silent)
[static,const]	Severity	**NoSeverity**	Specifies no particular severity (default)
[static,const]	Severity	**Warning**	Specifies warning severity (log with high priority, but do not stop)

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const Severity other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const Severity other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const Severity other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other)

Description: Compares an enum with an integer value

Error

Signature: *[static,const]* Severity **Error**

Description: Specifies error severity (preferred action is stop)

Python specific notes:
The object exposes a readable attribute 'Error'. This is the getter.

Info

Signature: *[static,const]* Severity **Info**

Description: Specifies info severity (print if requested, otherwise silent)

Python specific notes:
The object exposes a readable attribute 'Info'. This is the getter.

NoSeverity

Signature: *[static,const]* Severity **NoSeverity**

Description: Specifies no particular severity (default)

Python specific notes:
The object exposes a readable attribute 'NoSeverity'. This is the getter.

Warning

Signature: *[static,const]* Severity **Warning**

Description: Specifies warning severity (log with high priority, but do not stop)

Python specific notes:
The object exposes a readable attribute 'Warning'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Severity other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Severity ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:

	This method is also available as 'hash(object)'.
inspect	**Signature**: *[const]* string **inspect** **Description**: Converts an enum to a visual string **Python specific notes:** This method is also available as 'repr(object)'.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**(1) Signature**: *[static]* new Severity ptr **new** (int i) **Description**: Creates an enum from an integer value **Python specific notes:** This method is the default initializer of the object. **(2) Signature**: *[static]* new Severity ptr **new** (string s) **Description**: Creates an enum from a string value **Python specific notes:** This method is the default initializer of the object.
to_i	**Signature**: *[const]* int **to_i** **Description**: Gets the integer value from the enum **Python specific notes:** This method is also available as 'int(object)'.
to_s	**Signature**: *[const]* string **to_s** **Description**: Gets the symbolic string from an enum **Python specific notes:** This method is also available as 'str(object)'.

4.64. API reference - Class Manager

Notation used in Ruby API documentation

Module: db

Description: A transaction manager class

Manager objects control layout and potentially other objects in the layout database and queue operations to form transactions. A transaction is a sequence of operations that can be undone or redone.

In order to equip a layout object with undo/redo support, instantiate the layout object with a manager attached and embrace the operations to undo/redo with transaction/commit calls.

The use of transactions is subject to certain constraints, i.e. transacted sequences may not be mixed with non-transacted ones.

This class has been introduced in version 0.19.

Public constructors

new Manager ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Manager other)	Assigns another object to self
	void	**commit**		Close a transaction.
[const]	new Manager ptr	**dup**		Creates a copy of self
[const]	bool	**has_redo?**		Determine if a transaction is available for 'redo'
[const]	bool	**has_undo?**		Determine if a transaction is available for 'undo'
	void	**redo**		Redo the next available transaction
	unsigned long	**transaction**	(string description)	Begin a transaction
	unsigned long	**transaction**	(string description,	Begin a joined transaction

		unsigned long join_with)	
[const]	string	**transaction_for_redo**	Return the description of the next transaction for 'redo'
[const]	string	**transaction_for_undo**	Return the description of the next transaction for 'undo'
	void	**undo**	Undo the current transaction

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Manager other)

Description: Assigns another object to self

commit

Signature: void **commit**

Description: Close a transaction.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Manager ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

has_redo?

Signature: *[const]* bool **has_redo?**

Description: Determine if a transaction is available for 'redo'

Returns: True, if a transaction is available.

has_undo?

Signature: *[const]* bool **has_undo?**

Description: Determine if a transaction is available for 'undo'

Returns: True, if a transaction is available.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new Manager ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

redo

Signature: void **redo**

Description: Redo the next available transaction

The next transaction is redone with this method. The 'has_redo' method can be used to determine whether there are transactions to undo.

transaction

(1) Signature: unsigned long **transaction** (string description)

Description: Begin a transaction

description: The description for this transaction.

Returns: The ID of the transaction (can be used to join other transactions with this one)

This call will open a new transaction. A transaction consists of a set of operations issued with the 'queue' method. A transaction is closed with the 'commit' method.

(2) Signature: unsigned long **transaction** (string description, unsigned long join_with)

Description: Begin a joined transaction

description: The description for this transaction (ignored if joined).

description: The ID of the previous transaction.

Returns: The ID of the new transaction (can be used to join more)

This call will open a new transaction and join if with the previous transaction. The ID of the previous transaction must be equal to the ID given with 'join_with'.

This overload was introduced in version 0.22.

transaction_for_redo

Signature: *[const]* string **transaction_for_redo**

Description: Return the description of the next transaction for 'redo'

transaction_for_undo	**Signature**: *[const]* string **transaction_for_undo** **Description**: Return the description of the next transaction for 'undo'
undo	**Signature**: void **undo** **Description**: Undo the current transaction The current transaction is undone with this method. The 'has_undo' method can be used to determine whether there are transactions to undo.

4.65. API reference - Class Matrix2d

Notation used in Ruby API documentation

Module: db

Description: A 2d matrix object used mainly for representing rotation and shear transformations.

This object represents a 2x2 matrix. This matrix is used to implement affine transformations in the 2d space mainly. It can be decomposed into basic transformations: mirroring, rotation and shear. In that case, the assumed execution order of the basic transformations is mirroring at the x axis, rotation, magnification and shear.

The matrix is a generalization of the transformations and is of limited use in a layout database context. It is useful however to implement shear transformations on polygons, edges and polygon or edge collections.

This class was introduced in version 0.22.

Public constructors

new Matrix2d ptr	**new**		Create a new Matrix2d representing a unit transformation
new Matrix2d ptr	**new**	(double m)	Create a new Matrix2d representing an isotropic magnification
new Matrix2d ptr	**new**	(double mx, double my)	Create a new Matrix2d representing an anisotropic magnification
new Matrix2d ptr	**new**	(const DCplxTrans t)	Create a new Matrix2d from the given complex transformation@param t The transformation from which to create the matrix (not taking into account the displacement)
new Matrix2d ptr	**new**	(double m11, double m12, double m21, double m22)	Create a new Matrix2d from the four coefficients

Public methods

[const]	DPoint	*	(const DPoint p)	Transforms a point with this matrix.
[const]	DVector	*	(const DVector v)	Transforms a vector with this matrix.
[const]	DEdge	*	(const DEdge e)	Transforms an edge with this matrix.
[const]	DBox	*	(const DBox box)	Transforms a box with this matrix.
[const]	DSimplePolygon	*	(const DSimplePolygon p)	Transforms a simple polygon with this matrix.
[const]	DPolygon	*	(const DPolygon p)	Transforms a polygon with this matrix.

[const]	Matrix2d	*****	(const Matrix2d m)	Product of two matrices.
[const]	Matrix2d	**+**	(const Matrix2d m)	Sum of two matrices.
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	double	**angle**		Returns the rotation angle of the rotation component of this matrix.
	void	**assign**	(const Matrix2d other)	Assigns another object to self
[const]	DCplxTrans	**cplx_trans**		Converts this matrix to a complex transformation (if possible).
[const]	new Matrix2d ptr	**dup**		Creates a copy of self
[const]	Matrix2d	**inverted**		The inverse of this matrix.
[const]	bool	**is_mirror?**		Returns the mirror flag of this matrix.
[const]	double	**m**	(int i, int j)	Gets the m coefficient with the given index.
[const]	double	**m11**		Gets the m11 coefficient.
[const]	double	**m12**		Gets the m12 coefficient.
[const]	double	**m21**		Gets the m21 coefficient.
[const]	double	**m22**		Gets the m22 coefficient.
[const]	double	**mag_x**		Returns the x magnification of the magnification component of this matrix.
[const]	double	**mag_y**		Returns the y magnification of the magnification component of this matrix.
[const]	double	**shear_angle**		Returns the magnitude of the shear component of this matrix.
[const]	string	**to_s**		Convert the matrix to a string.

[const]	DPoint	**trans**	(const DPoint p)	Transforms a point with this matrix.

Public static methods and constants

	new Matrix2d ptr	**newc**	(double mag, double rotation, bool mirror)	Create a new Matrix2d representing an isotropic magnification, rotation and mirroring
	new Matrix2d ptr	**newc**	(double shear, double mx, double my, double rotation, bool mirror)	Create a new Matrix2d representing a shear, anisotropic magnification, rotation and mirroring

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

*****

(1) Signature: *[const]* DPoint ***** (const DPoint p)

Description: Transforms a point with this matrix.

p: The point to transform.

Returns: The transformed point

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* DVector ***** (const DVector v)

Description: Transforms a vector with this matrix.

v: The vector to transform.

Returns: The transformed vector

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* DEdge ***** (const DEdge e)

Description: Transforms an edge with this matrix.

e: The edge to transform.

Returns: The transformed edge

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* DBox ***** (const DBox box)

Description: Transforms a box with this matrix.

box:	The box to transform.
Returns:	The transformed box

Please note that the box remains a box, even though the matrix supports shear and rotation. The returned box will be the bounding box of the sheared and rotated rectangle.

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* DSimplePolygon ***** (const DSimplePolygon p)

Description: Transforms a simple polygon with this matrix.

p:	The simple polygon to transform.
Returns:	The transformed simple polygon

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* DPolygon ***** (const DPolygon p)

Description: Transforms a polygon with this matrix.

p:	The polygon to transform.
Returns:	The transformed polygon

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* Matrix2d ***** (const Matrix2d m)

Description: Product of two matrices.

m:	The other matrix.
Returns:	The matrix product self*m

Python specific notes:
This method also implements '__rmul__'.

+

Signature: *[const]* Matrix2d **+** (const Matrix2d m)

Description: Sum of two matrices.

m:	The other matrix.
Returns:	The (element-wise) sum of self+m

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

angle

Signature: *[const]* double **angle**

Description: Returns the rotation angle of the rotation component of this matrix.

Returns: The angle in degree.

The matrix is decomposed into basic transformations assuming an execution order of mirroring at the x axis, rotation, magnification and shear.

assign

Signature: void **assign** (const Matrix2d other)

Description: Assigns another object to self

cplx_trans

Signature: *[const]* DCplxTrans **cplx_trans**

Description: Converts this matrix to a complex transformation (if possible).

Returns: The complex transformation.

This method is successful only if the matrix does not contain shear components and the magnification must be isotropic.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Matrix2d ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

inverted

Signature: *[const]* Matrix2d **inverted**

Description: The inverse of this matrix.

Returns: The inverse of this matrix

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_mirror?

Signature: *[const]* bool **is_mirror?**

Description: Returns the mirror flag of this matrix.

Returns: True if this matrix has a mirror component.

The matrix is decomposed into basic transformations assuming an execution order of mirroring at the x axis, rotation, magnification and shear.

m

Signature: *[const]* double **m** (int i, int j)

Description: Gets the m coefficient with the given index.

Returns: The coefficient [i,j]

m11

Signature: *[const]* double **m11**

Description: Gets the m11 coefficient.

Returns: The value of the m11 coefficient

m12

Signature: *[const]* double **m12**

Description: Gets the m12 coefficient.

Returns: The value of the m12 coefficient

m21

Signature: *[const]* double **m21**

Description: Gets the m21 coefficient.

Returns: The value of the m21 coefficient

m22

Signature: *[const]* double **m22**

Description: Gets the m22 coefficient.

Returns: The value of the m22 coefficient

mag_x

Signature: *[const]* double **mag_x**

Description: Returns the x magnification of the magnification component of this matrix.

Returns: The magnification factor.

The matrix is decomposed into basic transformations assuming an execution order of mirroring at the x axis, magnification, shear and rotation.

mag_y

Signature: *[const]* double **mag_y**

Description: Returns the y magnification of the magnification component of this matrix.

Returns: The magnification factor.

The matrix is decomposed into basic transformations assuming an execution order of mirroring at the x axis, magnification, shear and rotation.

new

(1) Signature: *[static]* new Matrix2d ptr **new**

Description: Create a new Matrix2d representing a unit transformation

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Matrix2d ptr **new** (double m)

Description: Create a new Matrix2d representing an isotropic magnification

m: The magnification

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Matrix2d ptr **new** (double mx, double my)

Description: Create a new Matrix2d representing an anisotropic magnification

mx: The magnification in x direction

my: The magnification in y direction

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Matrix2d ptr **new** (const DCplxTrans t)

Description: Create a new Matrix2d from the given complex transformation@param t The transformation from which to create the matrix (not taking into account the displacement)

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new Matrix2d ptr **new** (double m11, double m12, double m21, double m22)

Description: Create a new Matrix2d from the four coefficients

Python specific notes:
This method is the default initializer of the object.

newc

(1) Signature: *[static]* new Matrix2d ptr **newc** (double mag, double rotation, bool mirror)

Description: Create a new Matrix2d representing an isotropic magnification, rotation and mirroring

mag:	The magnification in x direction
rotation:	The rotation angle (in degree)
mirror:	The mirror flag (at x axis)

This constructor is provided to construct a matrix similar to the complex transformation. This constructor is called 'newc' to distinguish it from the constructors taking matrix coefficients ('c' is for composite). The order of execution of the operations is mirror, magnification, rotation (as for complex transformations).

(2) Signature: *[static]* new Matrix2d ptr **newc** (double shear, double mx, double my, double rotation, bool mirror)

Description: Create a new Matrix2d representing a shear, anisotropic magnification, rotation and mirroring

shear:	The shear angle
mx:	The magnification in x direction
my:	The magnification in y direction
rotation:	The rotation angle (in degree)
mirror:	The mirror flag (at x axis)

The order of execution of the operations is mirror, magnification, shear and rotation. This constructor is called 'newc' to distinguish it from the constructor taking the four matrix coefficients ('c' is for composite).

shear_angle

Signature: *[const]* double **shear_angle**

Description: Returns the magnitude of the shear component of this matrix.

Returns:	The shear angle in degree.

The matrix is decomposed into basic transformations assuming an execution order of mirroring at the x axis, rotation, magnification and shear. The shear basic transformation will tilt the x axis towards the y axis and vice versa. The shear angle gives the tilt angle of the axes towards the other one. The possible range for this angle is -45 to 45 degree.

to_s

Signature: *[const]* string **to_s**

Description: Convert the matrix to a string.

Returns: The string representing this matrix

Python specific notes:
This method is also available as 'str(object)'.

trans

Signature: *[const]* DPoint **trans** (const DPoint p)

Description: Transforms a point with this matrix.

p: The point to transform.

Returns: The transformed point

Python specific notes:
This method also implements '__rmul__'.

4.66. API reference - Class IMatrix2d

Notation used in Ruby API documentation

Module: db

Description: A 2d matrix object used mainly for representing rotation and shear transformations (integer coordinate version).

This object represents a 2x2 matrix. This matrix is used to implement affine transformations in the 2d space mainly. It can be decomposed into basic transformations: mirroring, rotation and shear. In that case, the assumed execution order of the basic transformations is mirroring at the x axis, rotation, magnification and shear.

The integer variant was introduced in version 0.27.

Public constructors

new IMatrix2d ptr	**new**		Create a new Matrix2d representing a unit transformation
new IMatrix2d ptr	**new**	(double m)	Create a new Matrix2d representing an isotropic magnification
new IMatrix2d ptr	**new**	(double mx, double my)	Create a new Matrix2d representing an anisotropic magnification
new IMatrix2d ptr	**new**	(const DCplxTrans t)	Create a new Matrix2d from the given complex transformation@param t The transformation from which to create the matrix (not taking into account the displacement)
new IMatrix2d ptr	**new**	(double m11, double m12, double m21, double m22)	Create a new Matrix2d from the four coefficients

Public methods

[const]	Point	*	(const Point p)	Transforms a point with this matrix.
[const]	Vector	*	(const Vector v)	Transforms a vector with this matrix.
[const]	Edge	*	(const Edge e)	Transforms an edge with this matrix.
[const]	Box	*	(const Box box)	Transforms a box with this matrix.
[const]	SimplePolygon	*	(const SimplePolygon p)	Transforms a simple polygon with this matrix.
[const]	Polygon	*	(const Polygon p)	Transforms a polygon with this matrix.
[const]	IMatrix2d	*	(const IMatrix2d m)	Product of two matrices.

[const]	IMatrix2d	**+**	(const IMatrix2d m)	Sum of two matrices.
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	double	**angle**		Returns the rotation angle of the rotation component of this matrix.
	void	**assign**	(const IMatrix2d other)	Assigns another object to self
[const]	ICplxTrans	**cplx_trans**		Converts this matrix to a complex transformation (if possible).
[const]	new IMatrix2d ptr	**dup**		Creates a copy of self
[const]	IMatrix2d	**inverted**		The inverse of this matrix.
[const]	bool	**is_mirror?**		Returns the mirror flag of this matrix.
[const]	double	**m**	(int i, int j)	Gets the m coefficient with the given index.
[const]	double	**m11**		Gets the m11 coefficient.
[const]	double	**m12**		Gets the m12 coefficient.
[const]	double	**m21**		Gets the m21 coefficient.
[const]	double	**m22**		Gets the m22 coefficient.
[const]	double	**mag_x**		Returns the x magnification of the magnification component of this matrix.
[const]	double	**mag_y**		Returns the y magnification of the magnification component of this matrix.
[const]	double	**shear_angle**		Returns the magnitude of the shear component of this matrix.
[const]	string	**to_s**		Convert the matrix to a string.
[const]	Point	**trans**	(const Point p)	Transforms a point with this matrix.

Public static methods and constants

	new IMatrix2d ptr	**newc**	(double mag, double rotation, bool mirror)	Create a new Matrix2d representing an isotropic magnification, rotation and mirroring
	new IMatrix2d ptr	**newc**	(double shear, double mx, double my, double rotation, bool mirror)	Create a new Matrix2d representing a shear, anisotropic magnification, rotation and mirroring

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

*****

(1) Signature: *[const]* Point ***** (const Point p)

Description: Transforms a point with this matrix.

p: The point to transform.

Returns: The transformed point

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* Vector ***** (const Vector v)

Description: Transforms a vector with this matrix.

v: The vector to transform.

Returns: The transformed vector

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* Edge ***** (const Edge e)

Description: Transforms an edge with this matrix.

e: The edge to transform.

Returns: The transformed edge

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* Box ***** (const Box box)

Description: Transforms a box with this matrix.

box: The box to transform.

Returns: The transformed box

Please note that the box remains a box, even though the matrix supports shear and rotation. The returned box will be the bounding box of the sheared and rotated rectangle.

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* SimplePolygon ***** (const SimplePolygon p)

Description: Transforms a simple polygon with this matrix.

p: The simple polygon to transform.

Returns: The transformed simple polygon

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* Polygon ***** (const Polygon p)

Description: Transforms a polygon with this matrix.

p: The polygon to transform.

Returns: The transformed polygon

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* IMatrix2d ***** (const IMatrix2d m)

Description: Product of two matrices.

m: The other matrix.

Returns: The matrix product self*m

Python specific notes:
This method also implements '__rmul__'.

+

Signature: *[const]* IMatrix2d **+** (const IMatrix2d m)

Description: Sum of two matrices.

m: The other matrix.

Returns: The (element-wise) sum of self+m

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

angle

Signature: *[const]* double **angle**

Description: Returns the rotation angle of the rotation component of this matrix.

Returns: The angle in degree.

The matrix is decomposed into basic transformations assuming an execution order of mirroring at the x axis, rotation, magnification and shear.

assign

Signature: void **assign** (const IMatrix2d other)

Description: Assigns another object to self

cplx_trans

Signature: *[const]* ICplxTrans **cplx_trans**

Description: Converts this matrix to a complex transformation (if possible).

Returns: The complex transformation.

This method is successful only if the matrix does not contain shear components and the magnification must be isotropic.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new IMatrix2d ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

inverted

Signature: *[const]* IMatrix2d **inverted**

Description: The inverse of this matrix.

Returns: The inverse of this matrix

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_mirror?

Signature: *[const]* bool **is_mirror?**

Description: Returns the mirror flag of this matrix.

Returns: True if this matrix has a mirror component.

The matrix is decomposed into basic transformations assuming an execution order of mirroring at the x axis, rotation, magnification and shear.

m

Signature: *[const]* double **m** (int i, int j)

Description: Gets the m coefficient with the given index.

Returns: The coefficient [i,j]

m11

Signature: *[const]* double **m11**

Description: Gets the m11 coefficient.

Returns: The value of the m11 coefficient

m12

Signature: *[const]* double **m12**

Description: Gets the m12 coefficient.

Returns: The value of the m12 coefficient

m21

Signature: *[const]* double **m21**

Description: Gets the m21 coefficient.

Returns: The value of the m21 coefficient

m22

Signature: *[const]* double **m22**

Description: Gets the m22 coefficient.

Returns: The value of the m22 coefficient

mag_x

Signature: *[const]* double **mag_x**

Description: Returns the x magnification of the magnification component of this matrix.

Returns: The magnification factor.

The matrix is decomposed into basic transformations assuming an execution order of mirroring at the x axis, magnification, shear and rotation.

mag_y

Signature: *[const]* double **mag_y**

Description: Returns the y magnification of the magnification component of this matrix.

Returns: The magnification factor.

The matrix is decomposed into basic transformations assuming an execution order of mirroring at the x axis, magnification, shear and rotation.

new

(1) Signature: *[static]* new IMatrix2d ptr **new**

Description: Create a new Matrix2d representing a unit transformation

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new IMatrix2d ptr **new** (double m)

Description: Create a new Matrix2d representing an isotropic magnification

m: The magnification

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new IMatrix2d ptr **new** (double mx, double my)

Description: Create a new Matrix2d representing an anisotropic magnification

mx: The magnification in x direction

my: The magnification in y direction

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new IMatrix2d ptr **new** (const DCplxTrans t)

Description: Create a new Matrix2d from the given complex transformation@param t The transformation from which to create the matrix (not taking into account the displacement)

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new IMatrix2d ptr **new** (double m11, double m12, double m21, double m22)

Description: Create a new Matrix2d from the four coefficients

Python specific notes:
This method is the default initializer of the object.

newc

(1) Signature: *[static]* new IMatrix2d ptr **newc** (double mag, double rotation, bool mirror)

Description: Create a new Matrix2d representing an isotropic magnification, rotation and mirroring

mag:	The magnification in x direction
rotation:	The rotation angle (in degree)
mirror:	The mirror flag (at x axis)

This constructor is provided to construct a matrix similar to the complex transformation. This constructor is called 'newc' to distinguish it from the constructors taking matrix coefficients ('c' is for composite). The order of execution of the operations is mirror, magnification, rotation (as for complex transformations).

(2) Signature: *[static]* new IMatrix2d ptr **newc** (double shear, double mx, double my, double rotation, bool mirror)

Description: Create a new Matrix2d representing a shear, anisotropic magnification, rotation and mirroring

shear:	The shear angle
mx:	The magnification in x direction
my:	The magnification in y direction
rotation:	The rotation angle (in degree)
mirror:	The mirror flag (at x axis)

The order of execution of the operations is mirror, magnification, shear and rotation. This constructor is called 'newc' to distinguish it from the constructor taking the four matrix coefficients ('c' is for composite).

shear_angle

Signature: *[const]* double **shear_angle**

Description: Returns the magnitude of the shear component of this matrix.

Returns:	The shear angle in degree.

The matrix is decomposed into basic transformations assuming an execution order of mirroring at the x axis, rotation, magnification and shear. The shear basic transformation will tilt the x axis towards the y axis and vice versa. The shear angle gives the tilt angle of the axes towards the other one. The possible range for this angle is -45 to 45 degree.

to_s

Signature: *[const]* string **to_s**

Description: Convert the matrix to a string.

Returns:	The string representing this matrix

Python specific notes:

This method is also available as 'str(object)'.

trans

Signature: *[const]* Point **trans** (const Point p)

Description: Transforms a point with this matrix.

p:	The point to transform.
Returns:	The transformed point

Python specific notes:
This method also implements '__rmul__'.

4.67. API reference - Class Matrix3d

Notation used in Ruby API documentation

Module: db

Description: A 3d matrix object used mainly for representing rotation, shear, displacement and perspective transformations.

This object represents a 3x3 matrix. This matrix is used to implement generic geometrical transformations in the 2d space mainly. It can be decomposed into basic transformations: mirroring, rotation, shear, displacement and perspective distortion. In that case, the assumed execution order of the basic transformations is mirroring at the x axis, rotation, magnification, shear, displacement and perspective distortion.

This class was introduced in version 0.22.

Public constructors

new Matrix3d ptr	**new**		Create a new Matrix3d representing a unit transformation
new Matrix3d ptr	**new**	(double m)	Create a new Matrix3d representing a magnification
new Matrix3d ptr	**new**	(const DCplxTrans t)	Create a new Matrix3d from the given complex transformation@param t The transformation from which to create the matrix
new Matrix3d ptr	**new**	(double m11, double m12, double m21, double m22)	Create a new Matrix3d from the four coefficients of a Matrix2d
new Matrix3d ptr	**new**	(double m11, double m12, double m21, double m22, double dx, double dy)	Create a new Matrix3d from the four coefficients of a Matrix2d plus a displacement
new Matrix3d ptr	**new**	(double m11, double m12, double m13, double m21, double m22, double m23, double m31, double m32, double m33)	Create a new Matrix3d from the nine matrix coefficients

Public methods

[const]	Matrix3d	*	(const Matrix3d m)	Product of two matrices.
[const]	DPoint	*	(const DPoint p)	Transforms a point with this matrix.
[const]	DVector	*	(const DVector v)	Transforms a vector with this matrix.

[const]	DEdge	*****	(const DEdge e)	Transforms an edge with this matrix.
[const]	DBox	*****	(const DBox box)	Transforms a box with this matrix.
[const]	DSimplePolygon	*****	(const DSimplePolygon p)	Transforms a simple polygon with this matrix.
[const]	DPolygon	*****	(const DPolygon p)	Transforms a polygon with this matrix.
[const]	Matrix3d	**+**	(const Matrix3d m)	Sum of two matrices.
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**adjust**	(DPoint[] landmarks_before, DPoint[] landmarks_after, int flags, int fixed_point)	Adjust a 3d matrix to match the given set of landmarks
[const]	double	**angle**		Returns the rotation angle of the rotation component of this matrix.
	void	**assign**	(const Matrix3d other)	Assigns another object to self
[const]	DCplxTrans	**cplx_trans**		Converts this matrix to a complex transformation (if possible).
[const]	DVector	**disp**		Returns the displacement vector of this transformation.
[const]	new Matrix3d ptr	**dup**		Creates a copy of self
[const]	Matrix3d	**inverted**		The inverse of this matrix.
[const]	bool	**is_mirror?**		Returns the mirror flag of this matrix.
[const]	double	**m**	(int i, int j)	Gets the m coefficient with the given index.
[const]	double	**mag_x**		Returns the x magnification of the magnification component of this matrix.

[const]	double	**mag_y**		Returns the y magnification of the magnification component of this matrix.
[const]	double	**shear_angle**		Returns the magnitude of the shear component of this matrix.
[const]	string	**to_s**		Convert the matrix to a string.
[const]	DPoint	**trans**	(const DPoint p)	Transforms a point with this matrix.
[const]	double	**tx**	(double z)	Returns the perspective tilt angle tx.
[const]	double	**ty**	(double z)	Returns the perspective tilt angle ty.

Public static methods and constants

int	**AdjustAll**		Mode for adjust: currently equivalent to adjust_perspective
int	**AdjustDisplacement**		Mode for adjust: adjust displacement only
int	**AdjustMagnification**		Mode for adjust: adjust rotation, mirror option and magnification
int	**AdjustNone**		Mode for adjust: adjust nothing
int	**AdjustPerspective**		Mode for adjust: adjust whole matrix including perspective transformation
int	**AdjustRotation**		Mode for adjust: adjust rotation only
int	**AdjustRotationMirror**		Mode for adjust: adjust rotation and mirror option
int	**AdjustShear**		Mode for adjust: adjust rotation, mirror option, magnification and shear
new Matrix3d ptr	**newc**	(double mag, double rotation, bool mirrx)	Create a new Matrix3d representing a isotropic magnification, rotation and mirroring
new Matrix3d ptr	**newc**	(double shear, double mx, double my, double rotation, bool mirrx)	Create a new Matrix3d representing a shear, anisotropic magnification, rotation and mirroring
new Matrix3d ptr	**newc**	(const DVector u, double shear, double mx, double my, double rotation, bool mirrx)	Create a new Matrix3d representing a displacement, shear, anisotropic magnification, rotation and mirroring
new Matrix3d ptr	**newc**	(double tx, double ty, double z, const DVector u, double shear,	Create a new Matrix3d representing a perspective distortion, displacement, shear, anisotropic magnification, rotation and mirroring

double mx,
double my,
double rotation,
bool mirrx)

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

*****

(1) Signature: *[const]* Matrix3d ***** (const Matrix3d m)

Description: Product of two matrices.

m: The other matrix.

Returns: The matrix product self*m

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* DPoint ***** (const DPoint p)

Description: Transforms a point with this matrix.

p: The point to transform.

Returns: The transformed point

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* DVector ***** (const DVector v)

Description: Transforms a vector with this matrix.

v: The vector to transform.

Returns: The transformed vector

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* DEdge ***** (const DEdge e)

Description: Transforms an edge with this matrix.

e: The edge to transform.

Returns: The transformed edge

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* DBox ***** (const DBox box)

Description: Transforms a box with this matrix.

box: The box to transform.

Returns: The transformed box

Please note that the box remains a box, even though the matrix supports shear and rotation. The returned box will be the bounding box of the sheared and rotated rectangle.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* DSimplePolygon ***** (const DSimplePolygon p)

Description: Transforms a simple polygon with this matrix.

p: The simple polygon to transform.

Returns: The transformed simple polygon

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* DPolygon ***** (const DPolygon p)

Description: Transforms a polygon with this matrix.

p: The polygon to transform.

Returns: The transformed polygon

Python specific notes:
This method also implements '__rmul__'.

+

Signature: *[const]* Matrix3d **+** (const Matrix3d m)

Description: Sum of two matrices.

m: The other matrix.

Returns: The (element-wise) sum of self+m

AdjustAll

Signature: *[static]* int **AdjustAll**

Description: Mode for adjust: currently equivalent to adjust_perspective

Python specific notes:
The object exposes a readable attribute 'AdjustAll'. This is the getter.

AdjustDisplacement

Signature: *[static]* int **AdjustDisplacement**

Description: Mode for adjust: adjust displacement only

Python specific notes:
The object exposes a readable attribute 'AdjustDisplacement'. This is the getter.

AdjustMagnification

Signature: *[static]* int **AdjustMagnification**

Description: Mode for adjust: adjust rotation, mirror option and magnification

Python specific notes:
The object exposes a readable attribute 'AdjustMagnification'. This is the getter.

AdjustNone

Signature: *[static]* int **AdjustNone**

Description: Mode for adjust: adjust nothing

Python specific notes:

	The object exposes a readable attribute 'AdjustNone'. This is the getter.
AdjustPerspective	**Signature**: *[static]* int **AdjustPerspective** **Description**: Mode for adjust: adjust whole matrix including perspective transformation **Python specific notes:** The object exposes a readable attribute 'AdjustPerspective'. This is the getter.
AdjustRotation	**Signature**: *[static]* int **AdjustRotation** **Description**: Mode for adjust: adjust rotation only **Python specific notes:** The object exposes a readable attribute 'AdjustRotation'. This is the getter.
AdjustRotationMirror	**Signature**: *[static]* int **AdjustRotationMirror** **Description**: Mode for adjust: adjust rotation and mirror option **Python specific notes:** The object exposes a readable attribute 'AdjustRotationMirror'. This is the getter.
AdjustShear	**Signature**: *[static]* int **AdjustShear** **Description**: Mode for adjust: adjust rotation, mirror option, magnification and shear **Python specific notes:** The object exposes a readable attribute 'AdjustShear'. This is the getter.
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

adjust

Signature: void **adjust** (DPoint[] landmarks_before, DPoint[] landmarks_after, int flags, int fixed_point)

Description: Adjust a 3d matrix to match the given set of landmarks

landmarks_before:	The points before the transformation.
landmarks_after:	The points after the transformation.
mode:	Selects the adjustment mode. Must be one of the Adjust... constants.
fixed_point:	The index of the fixed point (one that is definitely mapped to the target) or -1 if there is none

This function tries to adjust the matrix such, that either the matrix is changed as little as possible (if few landmarks are given) or that the "after" landmarks will match as close as possible to the "before" landmarks (if the problem is overdetermined).

angle

Signature: *[const]* double **angle**

Description: Returns the rotation angle of the rotation component of this matrix.

Returns:	The angle in degree.

See the description of this class for details about the basic transformations.

assign

Signature: void **assign** (const Matrix3d other)

Description: Assigns another object to self

cplx_trans

Signature: *[const]* DCplxTrans **cplx_trans**

Description: Converts this matrix to a complex transformation (if possible).

Returns:	The complex transformation.

This method is successful only if the matrix does not contain shear or perspective distortion components and the magnification must be isotropic.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disp

Signature: *[const]* DVector **disp**

Description: Returns the displacement vector of this transformation.

Returns: The displacement vector.

Starting with version 0.25 this method returns a vector type instead of a point.

dup

Signature: *[const]* new Matrix3d ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

inverted

Signature: *[const]* Matrix3d **inverted**

Description: The inverse of this matrix.

Returns: The inverse of this matrix

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_mirror?

Signature: *[const]* bool **is_mirror?**

Description: Returns the mirror flag of this matrix.

Returns: True if this matrix has a mirror component.

See the description of this class for details about the basic transformations.

m

Signature: *[const]* double **m** (int i, int j)

Description: Gets the m coefficient with the given index.

Returns: The coefficient [i,j]

mag_x

Signature: *[const]* double **mag_x**

Description: Returns the x magnification of the magnification component of this matrix.

Returns: The magnification factor.

mag_y

Signature: *[const]* double **mag_y**

Description: Returns the y magnification of the magnification component of this matrix.

Returns: The magnification factor.

new

(1) Signature: *[static]* new Matrix3d ptr **new**

Description: Create a new Matrix3d representing a unit transformation

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Matrix3d ptr **new** (double m)

Description: Create a new Matrix3d representing a magnification

m: The magnification

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Matrix3d ptr **new** (const DCplxTrans t)

Description: Create a new Matrix3d from the given complex transformation@param t The transformation from which to create the matrix

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Matrix3d ptr **new** (double m11, double m12, double m21, double m22)

Description: Create a new Matrix3d from the four coefficients of a Matrix2d

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new Matrix3d ptr **new** (double m11, double m12, double m21, double m22, double dx, double dy)

Description: Create a new Matrix3d from the four coefficients of a Matrix2d plus a displacement

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new Matrix3d ptr **new** (double m11, double m12, double m13, double m21, double m22, double m23, double m31, double m32, double m33)

Description: Create a new Matrix3d from the nine matrix coefficients

Python specific notes:
This method is the default initializer of the object.

newc

(1) Signature: *[static]* new Matrix3d ptr **newc** (double mag, double rotation, bool mirrx)

Description: Create a new Matrix3d representing a isotropic magnification, rotation and mirroring

mag:	The magnification
rotation:	The rotation angle (in degree)
mirrx:	The mirror flag (at x axis)

The order of execution of the operations is mirror, magnification and rotation. This constructor is called 'newc' to distinguish it from the constructors taking coefficients ('c' is for composite).

(2) Signature: *[static]* new Matrix3d ptr **newc** (double shear, double mx, double my, double rotation, bool mirrx)

Description: Create a new Matrix3d representing a shear, anisotropic magnification, rotation and mirroring

shear:	The shear angle
mx:	The magnification in x direction
mx:	The magnification in y direction
rotation:	The rotation angle (in degree)
mirrx:	The mirror flag (at x axis)

The order of execution of the operations is mirror, magnification, rotation and shear. This constructor is called 'newc' to distinguish it from the constructor taking the four matrix coefficients ('c' is for composite).

(3) Signature: *[static]* new Matrix3d ptr **newc** (const DVector u, double shear, double mx, double my, double rotation, bool mirrx)

Description: Create a new Matrix3d representing a displacement, shear, anisotropic magnification, rotation and mirroring

u:	The displacement
shear:	The shear angle
mx:	The magnification in x direction
mx:	The magnification in y direction
rotation:	The rotation angle (in degree)
mirrx:	The mirror flag (at x axis)

The order of execution of the operations is mirror, magnification, rotation, shear and displacement. This constructor is called 'newc' to distinguish it from the constructor taking the four matrix coefficients ('c' is for composite).

Starting with version 0.25 the displacement is of vector type.

(4) Signature: *[static]* new Matrix3d ptr **newc** (double tx, double ty, double z, const DVector u, double shear, double mx, double my, double rotation, bool mirrx)

Description: Create a new Matrix3d representing a perspective distortion, displacement, shear, anisotropic magnification, rotation and mirroring

tx:	The perspective tilt angle x (around the y axis)
ty:	The perspective tilt angle y (around the x axis)
z:	The observer distance at which the tilt angles are given
u:	The displacement
shear:	The shear angle
mx:	The magnification in x direction
mx:	The magnification in y direction

rotation: The rotation angle (in degree)

mirrx: The mirror flag (at x axis)

The order of execution of the operations is mirror, magnification, rotation, shear, perspective distortion and displacement. This constructor is called 'newc' to distinguish it from the constructor taking the four matrix coefficients ('c' is for composite).

The tx and ty parameters represent the perspective distortion. They denote a tilt of the xy plane around the y axis (tx) or the x axis (ty) in degree. The same effect is achieved for different tilt angles for different observer distances. Hence, the observer distance must be given at which the tilt angles are given. If the magnitude of the tilt angle is not important, z can be set to 1.

Starting with version 0.25 the displacement is of vector type.

shear_angle

Signature: *[const]* double **shear_angle**

Description: Returns the magnitude of the shear component of this matrix.

Returns: The shear angle in degree.

The shear basic transformation will tilt the x axis towards the y axis and vice versa. The shear angle gives the tilt angle of the axes towards the other one. The possible range for this angle is -45 to 45 degree.See the description of this class for details about the basic transformations.

to_s

Signature: *[const]* string **to_s**

Description: Convert the matrix to a string.

Returns: The string representing this matrix

Python specific notes:
This method is also available as 'str(object)'.

trans

Signature: *[const]* DPoint **trans** (const DPoint p)

Description: Transforms a point with this matrix.

p: The point to transform.

Returns: The transformed point

Python specific notes:
This method also implements '__rmul__'.

tx

Signature: *[const]* double **tx** (double z)

Description: Returns the perspective tilt angle tx.

z: The observer distance at which the tilt angle is computed.

Returns: The tilt angle tx.

The tx and ty parameters represent the perspective distortion. They denote a tilt of the xy plane around the y axis (tx) or the x axis (ty) in degree. The same effect is achieved for different tilt angles at different observer distances. Hence, the observer distance must be specified at which the tilt angle is computed. If the magnitude of the tilt angle is not important, z can be set to 1.

ty

Signature: *[const]* double **ty** (double z)

Description: Returns the perspective tilt angle ty.

z: The observer distance at which the tilt angle is computed.

Returns: The tilt angle ty.

The tx and ty parameters represent the perspective distortion. They denote a tilt of the xy plane around the y axis (tx) or the x axis (ty) in degree. The same effect is achieved for different tilt angles

at different observer distances. Hence, the observer distance must be specified at which the tilt angle is computed. If the magnitude of the tilt angle is not important, z can be set to 1.

4.68. API reference - Class IMatrix3d

Notation used in Ruby API documentation

Module: db

Description: A 3d matrix object used mainly for representing rotation, shear, displacement and perspective transformations (integer coordinate version).

This object represents a 3x3 matrix. This matrix is used to implement generic geometrical transformations in the 2d space mainly. It can be decomposed into basic transformations: mirroring, rotation, shear, displacement and perspective distortion. In that case, the assumed execution order of the basic transformations is mirroring at the x axis, rotation, magnification, shear, displacement and perspective distortion.

The integer variant was introduced in version 0.27.

Public constructors

new IMatrix3d ptr	**new**		Create a new Matrix3d representing a unit transformation
new IMatrix3d ptr	**new**	(double m)	Create a new Matrix3d representing a magnification
new IMatrix3d ptr	**new**	(const ICplxTrans t)	Create a new Matrix3d from the given complex transformation@param t The transformation from which to create the matrix
new IMatrix3d ptr	**new**	(double m11, double m12, double m21, double m22)	Create a new Matrix3d from the four coefficients of a Matrix2d
new IMatrix3d ptr	**new**	(double m11, double m12, double m21, double m22, double dx, double dy)	Create a new Matrix3d from the four coefficients of a Matrix2d plus a displacement
new IMatrix3d ptr	**new**	(double m11, double m12, double m13, double m21, double m22, double m23, double m31, double m32, double m33)	Create a new Matrix3d from the nine matrix coefficients

Public methods

[const]	IMatrix3d	*	(const IMatrix3d m)	Product of two matrices.
[const]	Point	*	(const Point p)	Transforms a point with this matrix.
[const]	Vector	*	(const Vector v)	Transforms a vector with this matrix.

[const]	Edge	*****	(const Edge e)	Transforms an edge with this matrix.
[const]	Box	*****	(const Box box)	Transforms a box with this matrix.
[const]	SimplePolygon	*****	(const SimplePolygon p)	Transforms a simple polygon with this matrix.
[const]	Polygon	*****	(const Polygon p)	Transforms a polygon with this matrix.
[const]	IMatrix3d	**+**	(const IMatrix3d m)	Sum of two matrices.
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	double	**angle**		Returns the rotation angle of the rotation component of this matrix.
	void	**assign**	(const IMatrix3d other)	Assigns another object to self
[const]	DCplxTrans	**cplx_trans**		Converts this matrix to a complex transformation (if possible).
[const]	Vector	**disp**		Returns the displacement vector of this transformation.
[const]	new IMatrix3d ptr	**dup**		Creates a copy of self
[const]	IMatrix3d	**inverted**		The inverse of this matrix.
[const]	bool	**is_mirror?**		Returns the mirror flag of this matrix.
[const]	double	**m**	(int i, int j)	Gets the m coefficient with the given index.
[const]	double	**mag_x**		Returns the x magnification of the magnification component of this matrix.
[const]	double	**mag_y**		Returns the y magnification of the magnification component of this matrix.

[const]	double	**shear_angle**		Returns the magnitude of the shear component of this matrix.
[const]	string	**to_s**		Convert the matrix to a string.
[const]	Point	**trans**	(const Point p)	Transforms a point with this matrix.
[const]	double	**tx**	(double z)	Returns the perspective tilt angle tx.
[const]	double	**ty**	(double z)	Returns the perspective tilt angle ty.

Public static methods and constants

new IMatrix3d ptr	**newc**	(double mag, double rotation, bool mirrx)	Create a new Matrix3d representing a isotropic magnification, rotation and mirroring
new IMatrix3d ptr	**newc**	(double shear, double mx, double my, double rotation, bool mirrx)	Create a new Matrix3d representing a shear, anisotropic magnification, rotation and mirroring
new IMatrix3d ptr	**newc**	(const Vector u, double shear, double mx, double my, double rotation, bool mirrx)	Create a new Matrix3d representing a displacement, shear, anisotropic magnification, rotation and mirroring
new IMatrix3d ptr	**newc**	(double tx, double ty, double z, const Vector u, double shear, double mx, double my, double rotation, bool mirrx)	Create a new Matrix3d representing a perspective distortion, displacement, shear, anisotropic magnification, rotation and mirroring

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

(1) Signature: *[const]* IMatrix3d ***** (const IMatrix3d m)

*****

Description: Product of two matrices.

m: The other matrix.

Returns: The matrix product self*m

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* Point ***** (const Point p)

Description: Transforms a point with this matrix.

p: The point to transform.

Returns: The transformed point

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* Vector ***** (const Vector v)

Description: Transforms a vector with this matrix.

v: The vector to transform.

Returns: The transformed vector

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* Edge ***** (const Edge e)

Description: Transforms an edge with this matrix.

e: The edge to transform.

Returns: The transformed edge

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* Box ***** (const Box box)

Description: Transforms a box with this matrix.

box: The box to transform.

Returns: The transformed box

Please note that the box remains a box, even though the matrix supports shear and rotation. The returned box will be the bounding box of the sheared and rotated rectangle.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* SimplePolygon ***** (const SimplePolygon p)

Description: Transforms a simple polygon with this matrix.

p: The simple polygon to transform.

Returns: The transformed simple polygon

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* Polygon ***** (const Polygon p)

Description: Transforms a polygon with this matrix.

p: The polygon to transform.

Returns: The transformed polygon

Python specific notes:
This method also implements '__rmul__'.

+

Signature: *[const]* IMatrix3d **+** (const IMatrix3d m)

Description: Sum of two matrices.

m: The other matrix.

Returns: The (element-wise) sum of self+m

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

angle

Signature: *[const]* double **angle**

Description: Returns the rotation angle of the rotation component of this matrix.

Returns: The angle in degree.

See the description of this class for details about the basic transformations.

assign

Signature: void **assign** (const IMatrix3d other)

Description: Assigns another object to self

cplx_trans

Signature: *[const]* DCplxTrans **cplx_trans**

Description: Converts this matrix to a complex transformation (if possible).

Returns: The complex transformation.

This method is successful only if the matrix does not contain shear or perspective distortion components and the magnification must be isotropic.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disp

Signature: *[const]* Vector **disp**

Description: Returns the displacement vector of this transformation.

Returns: The displacement vector.

Starting with version 0.25 this method returns a vector type instead of a point.

dup

Signature: *[const]* new IMatrix3d ptr **dup**

Description: Creates a copy of self

Python specific notes:

This method also implements '__copy__' and '__deepcopy__'.

inverted

Signature: *[const]* IMatrix3d **inverted**

Description: The inverse of this matrix.

Returns: The inverse of this matrix

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_mirror?

Signature: *[const]* bool **is_mirror?**

Description: Returns the mirror flag of this matrix.

Returns: True if this matrix has a mirror component.

See the description of this class for details about the basic transformations.

m

Signature: *[const]* double **m** (int i, int j)

Description: Gets the m coefficient with the given index.

Returns: The coefficient [i,j]

mag_x

Signature: *[const]* double **mag_x**

Description: Returns the x magnification of the magnification component of this matrix.

Returns: The magnification factor.

mag_y

Signature: *[const]* double **mag_y**

Description: Returns the y magnification of the magnification component of this matrix.

Returns: The magnification factor.

new

(1) Signature: *[static]* new IMatrix3d ptr **new**

Description: Create a new Matrix3d representing a unit transformation

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new IMatrix3d ptr **new** (double m)

Description: Create a new Matrix3d representing a magnification

m: The magnification

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new IMatrix3d ptr **new** (const ICplxTrans t)

Description: Create a new Matrix3d from the given complex transformation@param t The transformation from which to create the matrix

Python specific notes:

This method is the default initializer of the object.

(4) Signature: *[static]* new IMatrix3d ptr **new** (double m11, double m12, double m21, double m22)

Description: Create a new Matrix3d from the four coefficients of a Matrix2d

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new IMatrix3d ptr **new** (double m11, double m12, double m21, double m22, double dx, double dy)

Description: Create a new Matrix3d from the four coefficients of a Matrix2d plus a displacement

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new IMatrix3d ptr **new** (double m11, double m12, double m13, double m21, double m22, double m23, double m31, double m32, double m33)

Description: Create a new Matrix3d from the nine matrix coefficients

Python specific notes:
This method is the default initializer of the object.

newc

(1) Signature: *[static]* new IMatrix3d ptr **newc** (double mag, double rotation, bool mirrx)

Description: Create a new Matrix3d representing a isotropic magnification, rotation and mirroring

mag:	The magnification
rotation:	The rotation angle (in degree)
mirrx:	The mirror flag (at x axis)

The order of execution of the operations is mirror, magnification and rotation. This constructor is called 'newc' to distinguish it from the constructors taking coefficients ('c' is for composite).

(2) Signature: *[static]* new IMatrix3d ptr **newc** (double shear, double mx, double my, double rotation, bool mirrx)

Description: Create a new Matrix3d representing a shear, anisotropic magnification, rotation and mirroring

shear:	The shear angle
mx:	The magnification in x direction
mx:	The magnification in y direction
rotation:	The rotation angle (in degree)
mirrx:	The mirror flag (at x axis)

The order of execution of the operations is mirror, magnification, rotation and shear. This constructor is called 'newc' to distinguish it from the constructor taking the four matrix coefficients ('c' is for composite).

(3) Signature: *[static]* new IMatrix3d ptr **newc** (const Vector u, double shear, double mx, double my, double rotation, bool mirrx)

Description: Create a new Matrix3d representing a displacement, shear, anisotropic magnification, rotation and mirroring

u:	The displacement
shear:	The shear angle

mx:	The magnification in x direction
mx:	The magnification in y direction
rotation:	The rotation angle (in degree)
mirrx:	The mirror flag (at x axis)

The order of execution of the operations is mirror, magnification, rotation, shear and displacement. This constructor is called 'newc' to distinguish it from the constructor taking the four matrix coefficients ('c' is for composite).

Starting with version 0.25 the displacement is of vector type.

(4) Signature: *[static]* new IMatrix3d ptr **newc** (double tx, double ty, double z, const Vector u, double shear, double mx, double my, double rotation, bool mirrx)

Description: Create a new Matrix3d representing a perspective distortion, displacement, shear, anisotropic magnification, rotation and mirroring

tx:	The perspective tilt angle x (around the y axis)
ty:	The perspective tilt angle y (around the x axis)
z:	The observer distance at which the tilt angles are given
u:	The displacement
shear:	The shear angle
mx:	The magnification in x direction
mx:	The magnification in y direction
rotation:	The rotation angle (in degree)
mirrx:	The mirror flag (at x axis)

The order of execution of the operations is mirror, magnification, rotation, shear, perspective distortion and displacement. This constructor is called 'newc' to distinguish it from the constructor taking the four matrix coefficients ('c' is for composite).

The tx and ty parameters represent the perspective distortion. They denote a tilt of the xy plane around the y axis (tx) or the x axis (ty) in degree. The same effect is achieved for different tilt angles for different observer distances. Hence, the observer distance must be given at which the tilt angles are given. If the magnitude of the tilt angle is not important, z can be set to 1.

Starting with version 0.25 the displacement is of vector type.

shear_angle

Signature: *[const]* double **shear_angle**

Description: Returns the magnitude of the shear component of this matrix.

Returns:	The shear angle in degree.

The shear basic transformation will tilt the x axis towards the y axis and vice versa. The shear angle gives the tilt angle of the axes towards the other one. The possible range for this angle is -45 to 45 degree.See the description of this class for details about the basic transformations.

to_s

Signature: *[const]* string **to_s**

Description: Convert the matrix to a string.

Returns:	The string representing this matrix

Python specific notes:
This method is also available as 'str(object)'.

trans

Signature: *[const]* Point **trans** (const Point p)

Description: Transforms a point with this matrix.

p: The point to transform.

Returns: The transformed point

Python specific notes:
This method also implements '__rmul__'.

tx

Signature: *[const]* double **tx** (double z)

Description: Returns the perspective tilt angle tx.

z: The observer distance at which the tilt angle is computed.

Returns: The tilt angle tx.

The tx and ty parameters represent the perspective distortion. They denote a tilt of the xy plane around the y axis (tx) or the x axis (ty) in degree. The same effect is achieved for different tilt angles at different observer distances. Hence, the observer distance must be specified at which the tilt angle is computed. If the magnitude of the tilt angle is not important, z can be set to 1.

ty

Signature: *[const]* double **ty** (double z)

Description: Returns the perspective tilt angle ty.

z: The observer distance at which the tilt angle is computed.

Returns: The tilt angle ty.

The tx and ty parameters represent the perspective distortion. They denote a tilt of the xy plane around the y axis (tx) or the x axis (ty) in degree. The same effect is achieved for different tilt angles at different observer distances. Hence, the observer distance must be specified at which the tilt angle is computed. If the magnitude of the tilt angle is not important, z can be set to 1.

4.69. API reference - Class LayoutMetaInfo

Notation used in Ruby API documentation

Module: db

Description: A piece of layout meta information

Layout meta information is basically additional data that can be attached to a layout. Layout readers may generate meta information and some writers will add layout information to the layout object. Some writers will also read meta information to determine certain attributes.

Multiple layout meta information objects can be attached to one layout using Layout#add_meta_info. Meta information is identified by a unique name and carries a string value plus an optional description string. The description string is for information only and is not evaluated by code.

Meta information can be attached to the layout object and to cells. It is similar to user properties. The differences are:

- Meta information is stored differently in GDS and OASIS files using the context information added by KLayout to annotated PCell or library cells too. Hence meta information does not pollute the standard user properties space.
- The value of meta information can be complex serializable types such as lists, hashes and elementary objects such as Box or DBox. Scalar types include floats and booleans.
- Meta information keys are strings and are supported also for GDS which only accepts integer number keys for user properties.

Elementary (serializable) objects are: Box, DBox, Edge, DEdge, EdgePair, DEdgePair, EdgePairs, Edges, LayerProperties, Matrix2d, Matrix3d, Path, DPath, Point, DPoint, Polygon, DPolygon, SimplePolygon, DSimplePolygon, Region, Text, DText, Texts, Trans, DTrans, CplxTrans, ICplxTrans, DCplxTrans, VCplxTrans, Vector, DVector (list may not be complete).

KLayout itself also generates meta information with specific keys. For disambiguation, namespaces can be established by prefixing the key strings with some unique identifier in XML fashion, like a domain name - e.g. 'example.com:key'.

Note: only meta information marked with is_persisted? == true is stored in GDS or OASIS files. This is not the default setting, so you need to explicitly set that flag.

See also Layout#each_meta_info, Layout#meta_info_value, Layout#meta_info and Layout#remove_meta_info as well as the corresponding Cell methods.

An example of how to attach persisted meta information to a cell is here:

```
ly = RBA::Layout::new
c1 = ly.create_cell("C1")

mi = RBA::LayoutMetaInfo::new("the-answer", 42.0)
mi.persisted = true
c1.add_meta_info(mi)

# will now hold this piece of meta information attached to cell 'C1':
ly.write("to.gds")
```

This class has been introduced in version 0.25 and was extended in version 0.28.8.

Public constructors

new LayoutMetaInfo ptr	**new**	(string name, variant value, string description = , bool persisted = false)	Creates a layout meta info object

Public methods

void	**_create**	Ensures the C++ object is created

	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
	void	assign	(const LayoutMetaInf other)	Assigns another object to self
[const]	string	description		Gets the description of the layout meta info object
	void	description=	(string description)	Sets the description of the layout meta info object
[const]	new LayoutMetaInfo ptr	dup		Creates a copy of self
[const]	bool	is_persisted?		Gets a value indicating whether the meta information will be persisted
[const]	string	name		Gets the name of the layout meta info object
	void	name=	(string name)	Sets the name of the layout meta info object
	void	persisted=	(bool flag)	Sets a value indicating whether the meta information will be persisted
[const]	variant	value		Gets the value of the layout meta info object
	void	value=	(variant value)	Sets the value of the layout meta info object

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
assign	**Signature**: void **assign** (const LayoutMetaInfo other) **Description**: Assigns another object to self
create	**Signature**: void **create** **Description**: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

description

Signature: *[const]* string **description**

Description: Gets the description of the layout meta info object

Python specific notes:
The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string description)

Description: Sets the description of the layout meta info object

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new LayoutMetaInfo ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_persisted?

Signature: *[const]* bool **is_persisted?**

Description: Gets a value indicating whether the meta information will be persisted

This predicate was introduced in version 0.28.8.

Python specific notes:
The object exposes a readable attribute 'persisted'. This is the getter.

name

Signature: *[const]* string **name**

Description: Gets the name of the layout meta info object

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string name)

Description: Sets the name of the layout meta info object

Python specific notes:
The object exposes a writable attribute 'name'. This is the setter.

new

Signature: *[static]* new LayoutMetaInfo ptr **new** (string name, variant value, string description = , bool persisted = false)

Description: Creates a layout meta info object

name:	The name
value:	The value
description:	An optional description text
persisted:	If true, the meta information will be persisted in some file formats, like GDS2

The 'persisted' attribute has been introduced in version 0.28.8.

Python specific notes:
This method is the default initializer of the object.

persisted=

Signature: void **persisted=** (bool flag)

Description: Sets a value indicating whether the meta information will be persisted

This predicate was introduced in version 0.28.8.

Python specific notes:
The object exposes a writable attribute 'persisted'. This is the setter.

value

Signature: *[const]* variant **value**

Description: Gets the value of the layout meta info object

Python specific notes:
The object exposes a readable attribute 'value'. This is the getter.

value=

Signature: void **value=** (variant value)

Description: Sets the value of the layout meta info object

Python specific notes:
The object exposes a writable attribute 'value'. This is the setter.

4.70. API reference - Class Path

Notation used in Ruby API documentation

Module: db

Description: A path class

A path consists of an sequence of line segments forming the 'spine' of the path and a width. In addition, the starting point can be drawn back by a certain extent (the 'begin extension') and the end point can be pulled forward somewhat (by the 'end extension').

A path may have round ends for special purposes. In particular, a round-ended path with a single point can represent a circle. Round-ended paths should have being and end extensions equal to half the width. Non-round-ended paths with a single point are allowed but the definition of the resulting shape in not well defined and may differ in other tools.

See The Database API for more details about the database objects.

Public constructors

new Path ptr	**new**	(const DPath dpath)	Creates an integer coordinate path from a floating-point coordinate path
new Path ptr	**new**		Default constructor: creates an empty (invalid) path with width 0
new Path ptr	**new**	(Point[] pts, int width)	Constructor given the points of the path's spine and the width
new Path ptr	**new**	(Point[] pts, int width, int bgn_ext, int end_ext)	Constructor given the points of the path's spine, the width and the extensions
new Path ptr	**new**	(Point[] pts, int width, int bgn_ext, int end_ext, bool round)	Constructor given the points of the path's spine, the width, the extensions and the round end flag

Public methods

[const]	bool	**!=**	(const Path p)	Inequality test
[const]	Path	*****	(double f)	Scaling by some factor
[const]	bool	**<**	(const Path p)	Less operator
[const]	bool	**==**	(const Path p)	Equality test
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference

	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	long	area		Returns the approximate area of the path
	void	assign	(const Path other)	Assigns another object to self
[const]	Box	bbox		Returns the bounding box of the path
[const]	int	bgn_ext		Get the begin extension
	void	bgn_ext=	(int ext)	Set the begin extension
[const]	new Path ptr	dup		Creates a copy of self
[const,iter]	Point	each_point		Get the points that make up the path's spine
[const]	int	end_ext		Get the end extension
	void	end_ext=	(int ext)	Set the end extension
[const]	unsigned long	hash		Computes a hash value
[const]	bool	is_round?		Returns true, if the path has round ends
[const]	unsigned int	length		Returns the length of the path
	Path	move	(const Vector p)	Moves the path.
	Path	move	(int dx, int dy)	Moves the path.
[const]	Path	moved	(const Vector p)	Returns the moved path (does not change self)
[const]	Path	moved	(int dx, int dy)	Returns the moved path (does not change self)
[const]	unsigned long	num_points		Get the number of points
[const]	unsigned long	perimeter		Returns the approximate perimeter of the path
	void	points=	(Point[] p)	Set the points of the path
[const]	Polygon	polygon		Convert the path to a polygon
	void	round=	(bool round_ends_flag)	Set the 'round ends' flag
[const]	Path	round_corners	(double radius, int npoints)	Creates a new path whose corners are interpolated with circular bends

[const]	SimplePolygon	**simple_polygon**		Convert the path to a simple polygon
[const]	DPath	**to_dtype**	(double dbu = 1)	Converts the path to a floating-point coordinate path
[const]	string	**to_s**		Convert to a string
[const]	Path	**transformed**	(const ICplxTrans t)	Transform the path.
[const]	Path	**transformed**	(const Trans t)	Transform the path.
[const]	DPath	**transformed**	(const CplxTrans t)	Transform the path.
[const]	int	**width**		Get the width
	void	**width=**	(int w)	Set the width

Public static methods and constants

new Path ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new Path ptr	**from_dpath**	(const DPath dpath)	Use of this method is deprecated. Use new instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[static]	new Path ptr	**new_pw**	(Point[] pts, int width)	Use of this method is deprecated. Use new instead
[static]	new Path ptr	**new_pwx**	(Point[] pts, int width, int bgn_ext, int end_ext)	Use of this method is deprecated. Use new instead
[static]	new Path ptr	**new_pwxr**	(Point[] pts, int width, int bgn_ext, int end_ext, bool round)	Use of this method is deprecated. Use new instead
[const]	unsigned long	**points**		Use of this method is deprecated. Use num_points instead

[const]	DPath	**transformed_cplx**	(const CplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

!=

Signature: *[const]* bool **!=** (const Path p)

Description: Inequality test

p: The object to compare against

*****

Signature: *[const]* Path ***** (double f)

Description: Scaling by some factor

Returns the scaled object. All coordinates are multiplied with the given factor and if necessary rounded.

Python specific notes:
This method also implements '__rmul__'.

<

Signature: *[const]* bool **<** (const Path p)

Description: Less operator

p: The object to compare against

This operator is provided to establish some, not necessarily a certain sorting order

==

Signature: *[const]* bool **==** (const Path p)

Description: Equality test

p: The object to compare against

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

area

Signature: *[const]* long **area**

Description: Returns the approximate area of the path

This method returns the approximate value of the area. It is computed from the length times the width. end extensions are taken into account correctly, but not effects of the corner interpolation. This method was added in version 0.22.

assign

Signature: void **assign** (const Path other)

Description: Assigns another object to self

bbox

Signature: *[const]* Box **bbox**

Description: Returns the bounding box of the path

bgn_ext

Signature: *[const]* int **bgn_ext**

Description: Get the begin extension

Python specific notes:
The object exposes a readable attribute 'bgn_ext'. This is the getter.

bgn_ext=

Signature: void **bgn_ext=** (int ext)

Description: Set the begin extension

Python specific notes:
The object exposes a writable attribute 'bgn_ext'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Path ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_point

Signature: *[const,iter]* Point **each_point**

Description: Get the points that make up the path's spine

end_ext

Signature: *[const]* int **end_ext**

Description: Get the end extension

Python specific notes:
The object exposes a readable attribute 'end_ext'. This is the getter.

end_ext=

Signature: void **end_ext=** (int ext)

Description: Set the end extension

Python specific notes:
The object exposes a writable attribute 'end_ext'. This is the setter.

from_dpath

Signature: *[static]* new Path ptr **from_dpath** (const DPath dpath)

Description: Creates an integer coordinate path from a floating-point coordinate path

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dpath'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new Path ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given polygon. This method enables polygons as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_round?

Signature: *[const]* bool **is_round?**

Description: Returns true, if the path has round ends

Python specific notes:
The object exposes a readable attribute 'round'. This is the getter.

length

Signature: *[const]* unsigned int **length**

Description: Returns the length of the path

the length of the path is determined by summing the lengths of the segments and adding begin and end extensions. For round-ended paths the length of the paths between the tips of the ends.

This method was added in version 0.23.

move

(1) Signature: Path **move** (const Vector p)

Description: Moves the path.

p:	The distance to move the path.
Returns:	The moved path.

Moves the path by the given offset and returns the moved path. The path is overwritten.

(2) Signature: Path **move** (int dx, int dy)

Description: Moves the path.

dx:	The x distance to move the path.
dy:	The y distance to move the path.
Returns:	The moved path.

Moves the path by the given offset and returns the moved path. The path is overwritten.

This version has been added in version 0.23.

moved

(1) Signature: *[const]* Path **moved** (const Vector p)

Description: Returns the moved path (does not change self)

p:	The distance to move the path.
Returns:	The moved path.

Moves the path by the given offset and returns the moved path. The path is not modified.

(2) Signature: *[const]* Path **moved** (int dx, int dy)

Description: Returns the moved path (does not change self)

dx:	The x distance to move the path.
dy:	The y distance to move the path.
Returns:	The moved path.

Moves the path by the given offset and returns the moved path. The path is not modified.

This version has been added in version 0.23.

new

(1) Signature: *[static]* new Path ptr **new** (const DPath dpath)

Description: Creates an integer coordinate path from a floating-point coordinate path

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dpath'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Path ptr **new**

Description: Default constructor: creates an empty (invalid) path with width 0

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Path ptr **new** (Point[] pts, int width)

Description: Constructor given the points of the path's spine and the width

pts:	The points forming the spine of the path
width:	The width of the path

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Path ptr **new** (Point[] pts, int width, int bgn_ext, int end_ext)

Description: Constructor given the points of the path's spine, the width and the extensions

pts:	The points forming the spine of the path
width:	The width of the path
bgn_ext:	The begin extension of the path
end_ext:	The end extension of the path

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new Path ptr **new** (Point[] pts, int width, int bgn_ext, int end_ext, bool round)

Description: Constructor given the points of the path's spine, the width, the extensions and the round end flag

pts:	The points forming the spine of the path
width:	The width of the path
bgn_ext:	The begin extension of the path
end_ext:	The end extension of the path

round: If this flag is true, the path will get rounded ends

Python specific notes:
This method is the default initializer of the object.

new_pw

Signature: *[static]* new Path ptr **new_pw** (Point[] pts, int width)

Description: Constructor given the points of the path's spine and the width

pts: The points forming the spine of the path

width: The width of the path

Use of this method is deprecated. Use new instead

Python specific notes:
This method is the default initializer of the object.

new_pwx

Signature: *[static]* new Path ptr **new_pwx** (Point[] pts, int width, int bgn_ext, int end_ext)

Description: Constructor given the points of the path's spine, the width and the extensions

pts: The points forming the spine of the path

width: The width of the path

bgn_ext: The begin extension of the path

end_ext: The end extension of the path

Use of this method is deprecated. Use new instead

Python specific notes:
This method is the default initializer of the object.

new_pwxr

Signature: *[static]* new Path ptr **new_pwxr** (Point[] pts, int width, int bgn_ext, int end_ext, bool round)

Description: Constructor given the points of the path's spine, the width, the extensions and the round end flag

pts: The points forming the spine of the path

width: The width of the path

bgn_ext: The begin extension of the path

end_ext: The end extension of the path

round: If this flag is true, the path will get rounded ends

Use of this method is deprecated. Use new instead

Python specific notes:
This method is the default initializer of the object.

num_points

Signature: *[const]* unsigned long **num_points**

Description: Get the number of points

Python specific notes:
The object exposes a readable attribute 'points'. This is the getter.

perimeter

Signature: *[const]* unsigned long **perimeter**

Description: Returns the approximate perimeter of the path

This method returns the approximate value of the perimeter. It is computed from the length and the width. end extensions are taken into account correctly, but not effects of the corner interpolation. This method was added in version 0.24.4.

points

Signature: *[const]* unsigned long **points**

Description: Get the number of points

Use of this method is deprecated. Use num_points instead

Python specific notes:
The object exposes a readable attribute 'points'. This is the getter.

points=

Signature: void **points=** (Point[] p)

Description: Set the points of the path

p: An array of points to assign to the path's spine

Python specific notes:
The object exposes a writable attribute 'points'. This is the setter.

polygon

Signature: *[const]* Polygon **polygon**

Description: Convert the path to a polygon

The returned polygon is not guaranteed to be non-self overlapping. This may happen if the path overlaps itself or contains very short segments.

round=

Signature: void **round=** (bool round_ends_flag)

Description: Set the 'round ends' flag

A path with round ends show half circles at the ends, instead of square or rectangular ends. Paths with this flag set should use a begin and end extension of half the width (see bgn_ext and end_ext). The interpretation of such paths in other tools may differ otherwise.

Python specific notes:
The object exposes a writable attribute 'round'. This is the setter.

round_corners

Signature: *[const]* Path **round_corners** (double radius, int npoints)

Description: Creates a new path whose corners are interpolated with circular bends

radius: The radius of the bends

npoints: The number of points (per full circle) used for interpolating the bends

This method has been introduced in version 0.25.

simple_polygon

Signature: *[const]* SimplePolygon **simple_polygon**

Description: Convert the path to a simple polygon

The returned polygon is not guaranteed to be non-selfoverlapping. This may happen if the path overlaps itself or contains very short segments.

to_dtype

Signature: *[const]* DPath **to_dtype** (double dbu = 1)

Description: Converts the path to a floating-point coordinate path

The database unit can be specified to translate the integer-coordinate path into a floating-point coordinate path in micron units. The database unit is basically a scaling factor.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s**

Description: Convert to a string

Python specific notes:
This method is also available as 'str(object)'.

transformed

(1) Signature: *[const]* Path **transformed** (const ICplxTrans t)

Description: Transform the path.

t: The transformation to apply.

Returns: The transformed path (in this case an integer coordinate path).

Transforms the path with the given complex transformation. Does not modify the path but returns the transformed path.

This method has been introduced in version 0.18.

(2) Signature: *[const]* Path **transformed** (const Trans t)

Description: Transform the path.

t: The transformation to apply.

Returns: The transformed path.

Transforms the path with the given transformation. Does not modify the path but returns the transformed path.

(3) Signature: *[const]* DPath **transformed** (const CplxTrans t)

Description: Transform the path.

t: The transformation to apply.

Returns: The transformed path.

Transforms the path with the given complex transformation. Does not modify the path but returns the transformed path.

transformed_cplx

Signature: *[const]* DPath **transformed_cplx** (const CplxTrans t)

Description: Transform the path.

t: The transformation to apply.

Returns: The transformed path.

Use of this method is deprecated. Use transformed instead

Transforms the path with the given complex transformation. Does not modify the path but returns the transformed path.

width

Signature: *[const]* int **width**

Description: Get the width

Python specific notes:
The object exposes a readable attribute 'width'. This is the getter.

width=

Signature: void **width=** (int w)

Description: Set the width

Python specific notes:
The object exposes a writable attribute 'width'. This is the setter.

4.71. API reference - Class DPath

Notation used in Ruby API documentation

Module: db

Description: A path class

A path consists of an sequence of line segments forming the 'spine' of the path and a width. In addition, the starting point can be drawn back by a certain extent (the 'begin extension') and the end point can be pulled forward somewhat (by the 'end extension').

A path may have round ends for special purposes. In particular, a round-ended path with a single point can represent a circle. Round-ended paths should have being and end extensions equal to half the width. Non-round-ended paths with a single point are allowed but the definition of the resulting shape in not well defined and may differ in other tools.

See The Database API for more details about the database objects.

Public constructors

new DPath ptr	**new**	(const Path path)	Creates a floating-point coordinate path from an integer coordinate path
new DPath ptr	**new**		Default constructor: creates an empty (invalid) path with width 0
new DPath ptr	**new**	(DPoint[] pts, double width)	Constructor given the points of the path's spine and the width
new DPath ptr	**new**	(DPoint[] pts, double width, double bgn_ext, double end_ext)	Constructor given the points of the path's spine, the width and the extensions
new DPath ptr	**new**	(DPoint[] pts, double width, double bgn_ext, double end_ext, bool round)	Constructor given the points of the path's spine, the width, the extensions and the round end flag

Public methods

[const]	bool	**!=**	(const DPath p)	Inequality test
[const]	DPath	*****	(double f)	Scaling by some factor
[const]	bool	**<**	(const DPath p)	Less operator
[const]	bool	**==**	(const DPath p)	Equality test
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference

	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	double	area		Returns the approximate area of the path
	void	assign	(const DPath other)	Assigns another object to self
[const]	DBox	bbox		Returns the bounding box of the path
[const]	double	bgn_ext		Get the begin extension
	void	bgn_ext=	(double ext)	Set the begin extension
[const]	new DPath ptr	dup		Creates a copy of self
[const,iter]	DPoint	each_point		Get the points that make up the path's spine
[const]	double	end_ext		Get the end extension
	void	end_ext=	(double ext)	Set the end extension
[const]	unsigned long	hash		Computes a hash value
[const]	bool	is_round?		Returns true, if the path has round ends
[const]	double	length		Returns the length of the path
	DPath	move	(const DVector p)	Moves the path.
	DPath	move	(double dx, double dy)	Moves the path.
[const]	DPath	moved	(const DVector p)	Returns the moved path (does not change self)
[const]	DPath	moved	(double dx, double dy)	Returns the moved path (does not change self)
[const]	unsigned long	num_points		Get the number of points
[const]	double	perimeter		Returns the approximate perimeter of the path
	void	points=	(DPoint[] p)	Set the points of the path
[const]	DPolygon	polygon		Convert the path to a polygon
	void	round=	(bool round_ends_flag)	Set the 'round ends' flag
[const]	DPath	round_corners	(double radius, int npoints, double accuracy)	Creates a new path whose corners are interpolated with circular bends

[const]	DSimplePolygon	**simple_polygon**		Convert the path to a simple polygon
[const]	Path	**to_itype**	(double dbu = 1)	Converts the path to an integer coordinate path
[const]	string	**to_s**		Convert to a string
[const]	Path	**transformed**	(const VCplxTrans t)	Transforms the polygon with the given complex transformation
[const]	DPath	**transformed**	(const DTrans t)	Transform the path.
[const]	DPath	**transformed**	(const DCplxTrans t)	Transform the path.
[const]	double	**width**		Get the width
	void	**width=**	(double w)	Set the width

Public static methods and constants

new DPath ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new DPath ptr	**from_ipath**	(const Path path)	Use of this method is deprecated. Use new instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[static]	new DPath ptr	**new_pw**	(DPoint[] pts, double width)	Use of this method is deprecated. Use new instead
[static]	new DPath ptr	**new_pwx**	(DPoint[] pts, double width, double bgn_ext, double end_ext)	Use of this method is deprecated. Use new instead
[static]	new DPath ptr	**new_pwxr**	(DPoint[] pts, double width, double bgn_ext, double end_ext, bool round)	Use of this method is deprecated. Use new instead
[const]	unsigned long	**points**		Use of this method is deprecated. Use num_points instead

[const]	DPath	**transformed_cplx**	(const DCplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

!=

Signature: *[const]* bool **!=** (const DPath p)

Description: Inequality test

p: The object to compare against

*****

Signature: *[const]* DPath ***** (double f)

Description: Scaling by some factor

Returns the scaled object. All coordinates are multiplied with the given factor and if necessary rounded.

Python specific notes:
This method also implements '__rmul__'.

<

Signature: *[const]* bool **<** (const DPath p)

Description: Less operator

p: The object to compare against

This operator is provided to establish some, not necessarily a certain sorting order

==

Signature: *[const]* bool **==** (const DPath p)

Description: Equality test

p: The object to compare against

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

area

Signature: *[const]* double **area**

Description: Returns the approximate area of the path

This method returns the approximate value of the area. It is computed from the length times the width. end extensions are taken into account correctly, but not effects of the corner interpolation. This method was added in version 0.22.

assign

Signature: void **assign** (const DPath other)

Description: Assigns another object to self

bbox

Signature: *[const]* DBox **bbox**

Description: Returns the bounding box of the path

bgn_ext

Signature: *[const]* double **bgn_ext**

Description: Get the begin extension

Python specific notes:
The object exposes a readable attribute 'bgn_ext'. This is the getter.

bgn_ext=

Signature: void **bgn_ext=** (double ext)

Description: Set the begin extension

Python specific notes:
The object exposes a writable attribute 'bgn_ext'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

	Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
dup	**Signature**: *[const]* new DPath ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
each_point	**Signature**: *[const,iter]* DPoint **each_point** **Description**: Get the points that make up the path's spine
end_ext	**Signature**: *[const]* double **end_ext** **Description**: Get the end extension **Python specific notes:** The object exposes a readable attribute 'end_ext'. This is the getter.
end_ext=	**Signature**: void **end_ext=** (double ext) **Description**: Set the end extension **Python specific notes:** The object exposes a writable attribute 'end_ext'. This is the setter.
from_ipath	**Signature**: *[static]* new DPath ptr **from_ipath** (const Path path) **Description**: Creates a floating-point coordinate path from an integer coordinate path Use of this method is deprecated. Use new instead This constructor has been introduced in version 0.25 and replaces the previous static method 'from_ipath'. **Python specific notes:** This method is the default initializer of the object.
from_s	**Signature**: *[static]* new DPath ptr **from_s** (string s) **Description**: Creates an object from a string Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given polygon. This method enables polygons as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_round?

Signature: *[const]* bool **is_round?**

Description: Returns true, if the path has round ends

Python specific notes:
The object exposes a readable attribute 'round'. This is the getter.

length

Signature: *[const]* double **length**

Description: Returns the length of the path

the length of the path is determined by summing the lengths of the segments and adding begin and end extensions. For round-ended paths the length of the paths between the tips of the ends.

This method was added in version 0.23.

move

(1) Signature: DPath **move** (const DVector p)

Description: Moves the path.

p:	The distance to move the path.
Returns:	The moved path.

Moves the path by the given offset and returns the moved path. The path is overwritten.

(2) Signature: DPath **move** (double dx, double dy)

Description: Moves the path.

dx:	The x distance to move the path.
dy:	The y distance to move the path.
Returns:	The moved path.

Moves the path by the given offset and returns the moved path. The path is overwritten.

This version has been added in version 0.23.

moved

(1) Signature: *[const]* DPath **moved** (const DVector p)

Description: Returns the moved path (does not change self)

p:	The distance to move the path.
Returns:	The moved path.

Moves the path by the given offset and returns the moved path. The path is not modified.

(2) Signature: *[const]* DPath **moved** (double dx, double dy)

Description: Returns the moved path (does not change self)

dx:	The x distance to move the path.
dy:	The y distance to move the path.
Returns:	The moved path.

Moves the path by the given offset and returns the moved path. The path is not modified.

This version has been added in version 0.23.

new

(1) Signature: *[static]* new DPath ptr **new** (const Path path)

Description: Creates a floating-point coordinate path from an integer coordinate path

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_ipath'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DPath ptr **new**

Description: Default constructor: creates an empty (invalid) path with width 0

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new DPath ptr **new** (DPoint[] pts, double width)

Description: Constructor given the points of the path's spine and the width

pts:	The points forming the spine of the path
width:	The width of the path

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new DPath ptr **new** (DPoint[] pts, double width, double bgn_ext, double end_ext)

Description: Constructor given the points of the path's spine, the width and the extensions

pts:	The points forming the spine of the path
width:	The width of the path
bgn_ext:	The begin extension of the path
end_ext:	The end extension of the path

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new DPath ptr **new** (DPoint[] pts, double width, double bgn_ext, double end_ext, bool round)

Description: Constructor given the points of the path's spine, the width, the extensions and the round end flag

pts:	The points forming the spine of the path
width:	The width of the path

bgn_ext: The begin extension of the path

end_ext: The end extension of the path

round: If this flag is true, the path will get rounded ends

Python specific notes:
This method is the default initializer of the object.

new_pw

Signature: *[static]* new DPath ptr **new_pw** (DPoint[] pts, double width)

Description: Constructor given the points of the path's spine and the width

pts: The points forming the spine of the path

width: The width of the path

Use of this method is deprecated. Use new instead

Python specific notes:
This method is the default initializer of the object.

new_pwx

Signature: *[static]* new DPath ptr **new_pwx** (DPoint[] pts, double width, double bgn_ext, double end_ext)

Description: Constructor given the points of the path's spine, the width and the extensions

pts: The points forming the spine of the path

width: The width of the path

bgn_ext: The begin extension of the path

end_ext: The end extension of the path

Use of this method is deprecated. Use new instead

Python specific notes:
This method is the default initializer of the object.

new_pwxr

Signature: *[static]* new DPath ptr **new_pwxr** (DPoint[] pts, double width, double bgn_ext, double end_ext, bool round)

Description: Constructor given the points of the path's spine, the width, the extensions and the round end flag

pts: The points forming the spine of the path

width: The width of the path

bgn_ext: The begin extension of the path

end_ext: The end extension of the path

round: If this flag is true, the path will get rounded ends

Use of this method is deprecated. Use new instead

Python specific notes:
This method is the default initializer of the object.

num_points

Signature: *[const]* unsigned long **num_points**

Description: Get the number of points

Python specific notes:
The object exposes a readable attribute 'points'. This is the getter.

perimeter

Signature: *[const]* double **perimeter**

Description: Returns the approximate perimeter of the path

This method returns the approximate value of the perimeter. It is computed from the length and the width. end extensions are taken into account correctly, but not effects of the corner interpolation. This method was added in version 0.24.4.

points

Signature: *[const]* unsigned long **points**

Description: Get the number of points

Use of this method is deprecated. Use num_points instead

Python specific notes:
The object exposes a readable attribute 'points'. This is the getter.

points=

Signature: void **points=** (DPoint[] p)

Description: Set the points of the path

p: An array of points to assign to the path's spine

Python specific notes:
The object exposes a writable attribute 'points'. This is the setter.

polygon

Signature: *[const]* DPolygon **polygon**

Description: Convert the path to a polygon

The returned polygon is not guaranteed to be non-self overlapping. This may happen if the path overlaps itself or contains very short segments.

round=

Signature: void **round=** (bool round_ends_flag)

Description: Set the 'round ends' flag

A path with round ends show half circles at the ends, instead of square or rectangular ends. Paths with this flag set should use a begin and end extension of half the width (see bgn_ext and end_ext). The interpretation of such paths in other tools may differ otherwise.

Python specific notes:
The object exposes a writable attribute 'round'. This is the setter.

round_corners

Signature: *[const]* DPath **round_corners** (double radius, int npoints, double accuracy)

Description: Creates a new path whose corners are interpolated with circular bends

radius: The radius of the bends

npoints: The number of points (per full circle) used for interpolating the bends

accuracy: The numerical accuracy of the computation

The accuracy parameter controls the numerical resolution of the approximation process and should be in the order of half the database unit. This accuracy is used for suppressing redundant points and simplification of the resulting path.

This method has been introduced in version 0.25.

simple_polygon

Signature: *[const]* DSimplePolygon **simple_polygon**

Description: Convert the path to a simple polygon

The returned polygon is not guaranteed to be non-selfoverlapping. This may happen if the path overlaps itself or contains very short segments.

to_itype

Signature: *[const]* Path **to_itype** (double dbu = 1)

Description: Converts the path to an integer coordinate path

The database unit can be specified to translate the floating-point coordinate path in micron units to an integer-coordinate path in database units. The path's' coordinates will be divided by the database unit.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s**

Description: Convert to a string

Python specific notes:
This method is also available as 'str(object)'.

transformed

(1) Signature: *[const]* Path **transformed** (const VCplxTrans t)

Description: Transforms the polygon with the given complex transformation

t:	The magnifying transformation to apply
Returns:	The transformed path (in this case an integer coordinate path)

This method has been introduced in version 0.25.

(2) Signature: *[const]* DPath **transformed** (const DTrans t)

Description: Transform the path.

t:	The transformation to apply.
Returns:	The transformed path.

Transforms the path with the given transformation. Does not modify the path but returns the transformed path.

(3) Signature: *[const]* DPath **transformed** (const DCplxTrans t)

Description: Transform the path.

t:	The transformation to apply.
Returns:	The transformed path.

Transforms the path with the given complex transformation. Does not modify the path but returns the transformed path.

transformed_cplx

Signature: *[const]* DPath **transformed_cplx** (const DCplxTrans t)

Description: Transform the path.

t:	The transformation to apply.
Returns:	The transformed path.

Use of this method is deprecated. Use transformed instead

Transforms the path with the given complex transformation. Does not modify the path but returns the transformed path.

width

Signature: *[const]* double **width**

Description: Get the width

Python specific notes:
The object exposes a readable attribute 'width'. This is the getter.

width=

Signature: void **width=** (double w)

Description: Set the width

Python specific notes:
The object exposes a writable attribute 'width'. This is the setter.

4.72. API reference - Class DPoint

Notation used in Ruby API documentation

Module: db

Description: A point class with double (floating-point) coordinates

Points represent a coordinate in the two-dimensional coordinate space of layout. They are not geometrical objects by itself. But they are frequently used in the database API for various purposes. Other than the integer variant (Point), points with floating-point coordinates can represent fractions of a database unit.

See The Database API for more details about the database objects.

Public constructors

new DPoint ptr	**new**	(const Point point)	Creates a floating-point coordinate point from an integer coordinate point
new DPoint ptr	**new**		Default constructor: creates a point at 0,0
new DPoint ptr	**new**	(const DVector v)	Default constructor: creates a point at from an vector
new DPoint ptr	**new**	(double x, double y)	Constructor for a point from two coordinate values

Public methods

[const]	bool	**!=**	(const DPoint p)	Inequality test operator
[const]	DPoint	*****	(double f)	Scaling by some factor
	DPoint	***=**	(double f)	Scaling by some factor
[const]	DPoint	**+**	(const DVector v)	Adds a vector to a point
[const]	DVector	**-**	(const DPoint p)	Subtract one point from another
[const]	DPoint	**-**	(const DVector v)	Subtract one vector from a point
[const]	DPoint	**-@**		Compute the negative of a point
[const]	DPoint	***/**	(double d)	Division by some divisor
	DPoint	***/=**	(double d)	Division by some divisor
[const]	bool	**<**	(const DPoint p)	"less" comparison operator
[const]	bool	**==**	(const DPoint p)	Equality test operator
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference

	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	double	abs		The absolute value of the point (Euclidian distance to 0,0)
	void	assign	(const DPoint other)	Assigns another object to self
[const]	double	distance	(const DPoint d)	The Euclidian distance to another point
[const]	new DPoint ptr	dup		Creates a copy of self
[const]	unsigned long	hash		Computes a hash value
[const]	double	sq_abs		The square of the absolute value of the point (Euclidian distance to 0,0)
[const]	double	sq_distance	(const DPoint d)	The square Euclidian distance to another point
[const]	Point	to_itype	(double dbu = 1)	Converts the point to an integer coordinate point
[const]	string	to_s	(double dbu = 0)	String conversion.
[const]	DVector	to_v		Turns the point into a vector
[const]	double	x		Accessor to the x coordinate
	void	x=	(double coord)	Write accessor to the x coordinate
[const]	double	y		Accessor to the y coordinate
	void	y=	(double coord)	Write accessor to the y coordinate

Public static methods and constants

new DPoint ptr	from_s	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead

[static]	new DPoint ptr	**from_ipoint**	(const Point point)	Use of this method is deprecated. Use new instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const DPoint p)
Description: Inequality test operator

*****

Signature: *[const]* DPoint ***** (double f)
Description: Scaling by some factor

Returns the scaled object. All coordinates are multiplied with the given factor and if necessary rounded.

Python specific notes:
This method also implements '__rmul__'.

***=**

Signature: DPoint ***=** (double f)
Description: Scaling by some factor

Scales object in place. All coordinates are multiplied with the given factor and if necessary rounded.

+

Signature: *[const]* DPoint **+** (const DVector v)
Description: Adds a vector to a point

Adds vector v to self by adding the coordinates.
Starting with version 0.25, this method expects a vector argument.

-

(1) Signature: *[const]* DVector **-** (const DPoint p)
Description: Subtract one point from another

Subtract point p from self by subtracting the coordinates. This renders a vector.
Starting with version 0.25, this method renders a vector.

(2) Signature: *[const]* DPoint **-** (const DVector v)
Description: Subtract one vector from a point

Subtract vector v from from self by subtracting the coordinates. This renders a point.
This method has been added in version 0.27.

-@

Signature: *[const]* DPoint **-@**
Description: Compute the negative of a point

Returns a new point with -x, -y.
This method has been added in version 0.23.

/

Signature: *[const]* DPoint **/** (double d)

Description: Division by some divisor

Returns the scaled object. All coordinates are divided with the given divisor and if necessary rounded.

/=

Signature: DPoint **/=** (double d)

Description: Division by some divisor

Divides the object in place. All coordinates are divided with the given divisor and if necessary rounded.

<

Signature: *[const]* bool **<** (const DPoint p)

Description: "less" comparison operator

This operator is provided to establish a sorting order

==

Signature: *[const]* bool **==** (const DPoint p)

Description: Equality test operator

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

abs

Signature: *[const]* double **abs**

Description: The absolute value of the point (Euclidian distance to 0,0)

The returned value is 'sqrt(x*x+y*y)'.

This method has been introduced in version 0.23.

assign

Signature: void **assign** (const DPoint other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

distance

Signature: *[const]* double **distance** (const DPoint d)

Description: The Euclidian distance to another point

d: The other point to compute the distance to.

dup

Signature: *[const]* new DPoint ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_ipoint

Signature: *[static]* new DPoint ptr **from_ipoint** (const Point point)

Description: Creates a floating-point coordinate point from an integer coordinate point

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_ipoint'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new DPoint ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given point. This method enables points as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

(1) Signature: *[static]* new DPoint ptr **new** (const Point point)

Description: Creates a floating-point coordinate point from an integer coordinate point

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_ipoint'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DPoint ptr **new**

Description: Default constructor: creates a point at 0,0

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new DPoint ptr **new** (const DVector v)

Description: Default constructor: creates a point at from an vector

This constructor is equivalent to computing point(0,0)+v. This method has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new DPoint ptr **new** (double x, double y)

Description: Constructor for a point from two coordinate values

Python specific notes:
This method is the default initializer of the object.

sq_abs

Signature: *[const]* double **sq_abs**

Description: The square of the absolute value of the point (Euclidian distance to 0,0)

The returned value is 'x*x+y*y'.

This method has been introduced in version 0.23.

sq_distance

Signature: *[const]* double **sq_distance** (const DPoint d)

Description: The square Euclidian distance to another point

d: The other point to compute the distance to.

to_itype

Signature: *[const]* Point **to_itype** (double dbu = 1)

Description: Converts the point to an integer coordinate point

The database unit can be specified to translate the floating-point coordinate point in micron units to an integer-coordinate point in database units. The point's' coordinates will be divided by the database unit.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: String conversion.

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

to_v

Signature: *[const]* DVector **to_v**

Description: Turns the point into a vector

This method returns a vector representing the distance from (0,0) to the point.This method has been introduced in version 0.25.

x

Signature: *[const]* double **x**

Description: Accessor to the x coordinate

Python specific notes:
The object exposes a readable attribute 'x'. This is the getter.

x=

Signature: void **x=** (double coord)

Description: Write accessor to the x coordinate

Python specific notes:
The object exposes a writable attribute 'x'. This is the setter.

y

Signature: *[const]* double **y**

Description: Accessor to the y coordinate

Python specific notes:

The object exposes a readable attribute 'y'. This is the getter.

y=

Signature: void **y=** (double coord)

Description: Write accessor to the y coordinate

Python specific notes:
The object exposes a writable attribute 'y'. This is the setter.

4.73. API reference - Class Point

Notation used in Ruby API documentation

Module: db

Description: An integer point class

Points represent a coordinate in the two-dimensional coordinate space of layout. They are not geometrical objects by itself. But they are frequently used in the database API for various purposes.

See The Database API for more details about the database objects.

Public constructors

new Point ptr	**new**	(const DPoint dpoint)	Creates an integer coordinate point from a floating-point coordinate point
new Point ptr	**new**		Default constructor: creates a point at 0,0
new Point ptr	**new**	(const Vector v)	Default constructor: creates a point at from an vector
new Point ptr	**new**	(int x, int y)	Constructor for a point from two coordinate values

Public methods

[const]	bool	**!=**	(const Point p)	Inequality test operator
[const]	Point	*****	(double f)	Scaling by some factor
	Point	***=**	(double f)	Scaling by some factor
[const]	Point	**+**	(const Vector v)	Adds a vector to a point
[const]	Vector	**-**	(const Point p)	Subtract one point from another
[const]	Point	**-**	(const Vector v)	Subtract one vector from a point
[const]	Point	**-@**		Compute the negative of a point
[const]	Point	**/**	(double d)	Division by some divisor
	Point	**/=**	(double d)	Division by some divisor
[const]	bool	**<**	(const Point p)	"less" comparison operator
[const]	bool	**==**	(const Point p)	Equality test operator
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference

	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	double	abs		The absolute value of the point (Euclidian distance to 0,0)
	void	assign	(const Point other)	Assigns another object to self
[const]	double	distance	(const Point d)	The Euclidian distance to another point
[const]	new Point ptr	dup		Creates a copy of self
[const]	unsigned long	hash		Computes a hash value
[const]	double	sq_abs		The square of the absolute value of the point (Euclidian distance to 0,0)
[const]	double	sq_distance	(const Point d)	The square Euclidian distance to another point
[const]	DPoint	to_dtype	(double dbu = 1)	Converts the point to a floating-point coordinate point
[const]	string	to_s	(double dbu = 0)	String conversion.
[const]	Vector	to_v		Turns the point into a vector
[const]	int	x		Accessor to the x coordinate
	void	x=	(int coord)	Write accessor to the x coordinate
[const]	int	y		Accessor to the y coordinate
	void	y=	(int coord)	Write accessor to the y coordinate

Public static methods and constants

new Point ptr	from_s	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	create		Use of this method is deprecated. Use _create instead
	void	destroy		Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?		Use of this method is deprecated. Use _destroyed? instead
[static]	new Point ptr	from_dpoint	(const DPoint dpoint)	Use of this method is deprecated. Use new instead

[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const Point p)
Description: Inequality test operator

*****

Signature: *[const]* Point ***** (double f)
Description: Scaling by some factor

Returns the scaled object. All coordinates are multiplied with the given factor and if necessary rounded.

Python specific notes:
This method also implements '__rmul__'.

***=**

Signature: Point ***=** (double f)
Description: Scaling by some factor

Scales object in place. All coordinates are multiplied with the given factor and if necessary rounded.

+

Signature: *[const]* Point **+** (const Vector v)
Description: Adds a vector to a point

Adds vector v to self by adding the coordinates.
Starting with version 0.25, this method expects a vector argument.

-

(1) Signature: *[const]* Vector **-** (const Point p)
Description: Subtract one point from another

Subtract point p from self by subtracting the coordinates. This renders a vector.
Starting with version 0.25, this method renders a vector.

(2) Signature: *[const]* Point **-** (const Vector v)
Description: Subtract one vector from a point

Subtract vector v from from self by subtracting the coordinates. This renders a point.
This method has been added in version 0.27.

-@

Signature: *[const]* Point **-@**
Description: Compute the negative of a point

Returns a new point with -x, -y.
This method has been added in version 0.23.

/

Signature: *[const]* Point **/** (double d)
Description: Division by some divisor

Returns the scaled object. All coordinates are divided with the given divisor and if necessary rounded.

/=

Signature: Point **/=** (double d)

Description: Division by some divisor

Divides the object in place. All coordinates are divided with the given divisor and if necessary rounded.

<

Signature: *[const]* bool **<** (const Point p)

Description: "less" comparison operator

This operator is provided to establish a sorting order

==

Signature: *[const]* bool **==** (const Point p)

Description: Equality test operator

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn

the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

abs

Signature: *[const]* double **abs**

Description: The absolute value of the point (Euclidian distance to 0,0)

The returned value is 'sqrt(x*x+y*y)'.

This method has been introduced in version 0.23.

assign

Signature: void **assign** (const Point other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

distance

Signature: *[const]* double **distance** (const Point d)

Description: The Euclidian distance to another point

d: The other point to compute the distance to.

dup

Signature: *[const]* new Point ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_dpoint

Signature: *[static]* new Point ptr **from_dpoint** (const DPoint dpoint)

Description: Creates an integer coordinate point from a floating-point coordinate point

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dpoint'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new Point ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given point. This method enables points as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

(1) Signature: *[static]* new Point ptr **new** (const DPoint dpoint)

Description: Creates an integer coordinate point from a floating-point coordinate point

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dpoint'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Point ptr **new**

Description: Default constructor: creates a point at 0,0

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Point ptr **new** (const Vector v)

Description: Default constructor: creates a point at from an vector

This constructor is equivalent to computing point(0,0)+v. This method has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Point ptr **new** (int x, int y)

Description: Constructor for a point from two coordinate values

Python specific notes:
This method is the default initializer of the object.

sq_abs

Signature: *[const]* double **sq_abs**

Description: The square of the absolute value of the point (Euclidian distance to 0,0)

The returned value is 'x*x+y*y'.

This method has been introduced in version 0.23.

sq_distance

Signature: *[const]* double **sq_distance** (const Point d)

Description: The square Euclidian distance to another point

d: The other point to compute the distance to.

to_dtype

Signature: *[const]* DPoint **to_dtype** (double dbu = 1)

Description: Converts the point to a floating-point coordinate point

The database unit can be specified to translate the integer-coordinate point into a floating-point coordinate point in micron units. The database unit is basically a scaling factor.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: String conversion.

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

to_v

Signature: *[const]* Vector **to_v**

Description: Turns the point into a vector

This method returns a vector representing the distance from (0,0) to the point.This method has been introduced in version 0.25.

x

Signature: *[const]* int **x**

Description: Accessor to the x coordinate

Python specific notes:
The object exposes a readable attribute 'x'. This is the getter.

x=

Signature: void **x=** (int coord)

Description: Write accessor to the x coordinate

Python specific notes:
The object exposes a writable attribute 'x'. This is the setter.

y

Signature: *[const]* int **y**

Description: Accessor to the y coordinate

Python specific notes:
The object exposes a readable attribute 'y'. This is the getter.

y=

Signature: void **y=** (int coord)

Description: Write accessor to the y coordinate

Python specific notes:
The object exposes a writable attribute 'y'. This is the setter.

4.74. API reference - Class SimplePolygon

Notation used in Ruby API documentation

Module: db

Description: A simple polygon class

A simple polygon consists of an outer hull only. To support polygons with holes, use Polygon. The hull contour consists of several points. The point list is normalized such that the leftmost, lowest point is the first one. The orientation is normalized such that the orientation of the hull contour is clockwise.

It is in no way checked that the contours are not overlapping This must be ensured by the user of the object when filling the contours.

The SimplePolygon class stores coordinates in integer format. A class that stores floating-point coordinates is DSimplePolygon.

See The Database API for more details about the database objects.

Public constructors

new SimplePolygon ptr	**new**	(const DSimplePolygon dpolygon)	Creates an integer coordinate polygon from a floating-point coordinate polygon
new SimplePolygon ptr	**new**		Default constructor: creates an empty (invalid) polygon
new SimplePolygon ptr	**new**	(Point[] pts, bool raw = false)	Constructor given the points of the simple polygon
new SimplePolygon ptr	**new**	(const Box box)	Constructor converting a box to a polygon

Public methods

[const]	bool	**!=**	(const SimplePolygon p)	Returns a value indicating whether self is not equal to p
[const]	SimplePolygon	*****	(double f)	Scales the polygon by some factor
[const]	bool	**<**	(const SimplePolygon p)	Returns a value indicating whether self is less than p
[const]	bool	**==**	(const SimplePolygon p)	Returns a value indicating whether self is equal to p
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

[const]	long	**area**		Gets the area of the polygon
[const]	long	**area2**		Gets the double area of the polygon
	void	**assign**	(const SimplePolygon other)	Assigns another object to self
[const]	Box	**bbox**		Returns the bounding box of the simple polygon
[const]	SimplePolygon[]	**break**	(unsigned long max_vertex_count, double max_area_ratio)	Splits the polygon into parts with a maximum vertex count and area ratio
	void	**compress**	(bool remove_reflected)	Compressed the simple polygon.
[const]	new SimplePolygon ptr	**dup**		Creates a copy of self
[const,iter]	Edge	**each_edge**		Iterates over the edges that make up the simple polygon
[const,iter]	Point	**each_point**		Iterates over the points that make up the simple polygon
[const]	variant[]	**extract_rad**		Extracts the corner radii from a rounded polygon
[const]	unsigned long	**hash**		Computes a hash value
[const]	bool	**inside?**	(Point p)	Gets a value indicating whether the given point is inside the polygon
[const]	bool	**is_box?**		Returns a value indicating whether the polygon is a simple box.
[const]	bool	**is_empty?**		Returns a value indicating whether the polygon is empty
[const]	bool	**is_halfmanhattan?**		Returns a value indicating whether the polygon is half-manhattan
[const]	bool	**is_rectilinear?**		Returns a value indicating whether the polygon is rectilinear
[const]	Polygon	**minkowski_sum**	(const Edge e, bool resolve_holes)	Computes the Minkowski sum of a polygon and an edge
[const]	Polygon	**minkowski_sum**	(const SimplePolygon p, bool resolve_holes)	Computes the Minkowski sum of a polygon and a polygon
[const]	Polygon	**minkowski_sum**	(const Box b,	Computes the Minkowski sum of a polygon and a box

			bool resolve_holes)	
[const]	Polygon	**minkowski_sum**	(Point[] c, bool resolve_holes)	Computes the Minkowski sum of a polygon and a contour of points (a trace)
	SimplePolygon	**move**	(const Vector p)	Moves the simple polygon.
	SimplePolygon	**move**	(int x, int y)	Moves the polygon.
[const]	SimplePolygon	**moved**	(const Vector p)	Returns the moved simple polygon
[const]	SimplePolygon	**moved**	(int x, int y)	Returns the moved polygon (does not modify self)
[const]	unsigned long	**num_points**		Gets the number of points
[const]	unsigned long	**perimeter**		Gets the perimeter of the polygon
[const]	Point	**point**	(unsigned long p)	Gets a specific point of the contour@param p The index of the point to get
	void	**points=**	(Point[] pts)	Sets the points of the simple polygon
[const]	SimplePolygon	**round_corners**	(double rinner, double router, unsigned int n)	Rounds the corners of the polygon
	void	**set_points**	(Point[] pts, bool raw = false)	Sets the points of the simple polygon
[const]	SimplePolygon[]	**split**		Splits the polygon into two or more parts
[const]	DSimplePolygon	**to_dtype**	(double dbu = 1)	Converts the polygon to a floating-point coordinate polygon
[const]	string	**to_s**		Returns a string representing the polygon
[const]	bool	**touches?**	(const Box box)	Returns true, if the polygon touches the given box.
[const]	bool	**touches?**	(const Edge edge)	Returns true, if the polygon touches the given edge.
[const]	bool	**touches?**	(const Polygon polygon)	Returns true, if the polygon touches the other polygon.
[const]	bool	**touches?**	(const SimplePolygon simple_polygon)	Returns true, if the polygon touches the other polygon.
	SimplePolygon ptr	**transform**	(const ICplxTrans t)	Transforms the simple polygon with a complex transformation (in-place)
	SimplePolygon ptr	**transform**	(const Trans t)	Transforms the simple polygon (in-place)

[const]	SimplePolygon	**transformed**	(const ICplxTrans t)	Transforms the simple polygon.
[const]	SimplePolygon	**transformed**	(const Trans t)	Transforms the simple polygon.
[const]	DSimplePolygon	**transformed**	(const CplxTrans t)	Transforms the simple polygon.

Public static methods and constants

	new SimplePolygon ptr	**ellipse**	(const Box box, int n)	Creates a simple polygon approximating an ellipse
	new SimplePolygon ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new SimplePolygon ptr	**from_dpoly**	(const DSimplePolygon dpolygon)	Use of this method is deprecated. Use new instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	Polygon	**minkowsky_sum**	(const Edge e, bool resolve_holes)	Use of this method is deprecated. Use minkowski_sum instead
[const]	Polygon	**minkowsky_sum**	(const SimplePolygon p, bool resolve_holes)	Use of this method is deprecated. Use minkowski_sum instead
[const]	Polygon	**minkowsky_sum**	(const Box b, bool resolve_holes)	Use of this method is deprecated. Use minkowski_sum instead
[const]	Polygon	**minkowsky_sum**	(Point[] c, bool resolve_holes)	Use of this method is deprecated. Use minkowski_sum instead
[const]	DSimplePolygon	**transformed_cplx**	(const CplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

!=

Signature: *[const]* bool **!=** (const SimplePolygon p)

Description: Returns a value indicating whether self is not equal to p

p: The object to compare against

*****

Signature: *[const]* SimplePolygon ***** (double f)

Description: Scales the polygon by some factor

Returns the scaled object. All coordinates are multiplied with the given factor and if necessary rounded.

Python specific notes:
This method also implements '__rmul__'.

<

Signature: *[const]* bool **<** (const SimplePolygon p)

Description: Returns a value indicating whether self is less than p

p: The object to compare against

This operator is provided to establish some, not necessarily a certain sorting order

This method has been introduced in version 0.25.

==

Signature: *[const]* bool **==** (const SimplePolygon p)

Description: Returns a value indicating whether self is equal to p

p: The object to compare against

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

	Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
area	**Signature**: *[const]* long **area** **Description**: Gets the area of the polygon The area is correct only if the polygon is not self-overlapping and the polygon is oriented clockwise.
area2	**Signature**: *[const]* long **area2** **Description**: Gets the double area of the polygon This method is provided because the area for an integer-type polygon is a multiple of 1/2. Hence the double area can be expresses precisely as an integer for these types. This method has been introduced in version 0.26.1
assign	**Signature**: void **assign** (const SimplePolygon other) **Description**: Assigns another object to self
bbox	**Signature**: *[const]* Box **bbox** **Description**: Returns the bounding box of the simple polygon
break	**Signature**: *[const]* SimplePolygon[] **break** (unsigned long max_vertex_count, double max_area_ratio) **Description**: Splits the polygon into parts with a maximum vertex count and area ratio The area ratio is the ratio between the bounding box area and the polygon area. Higher values mean more 'skinny' polygons. This method will split the input polygon into pieces having a maximum of 'max_vertex_count' vertices and an area ratio less than 'max_area_ratio'. 'max_vertex_count' can be zero. In this case the limit is ignored. Also 'max_area_ratio' can be zero, in which case it is ignored as well. The method of splitting is unspecified. The algorithm will apply 'split' recursively until the parts satisfy the limits. This method has been introduced in version 0.29. **Python specific notes:** This attribute is available as 'break_' in Python.
compress	**Signature**: void **compress** (bool remove_reflected) **Description**: Compressed the simple polygon. **remove_reflected**: See description of the functionality.

This method removes redundant points from the polygon, such as points being on a line formed by two other points. If remove_reflected is true, points are also removed if the two adjacent edges form a spike.

This method was introduced in version 0.18.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new SimplePolygon ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_edge

Signature: *[const,iter]* Edge **each_edge**

Description: Iterates over the edges that make up the simple polygon

each_point

Signature: *[const,iter]* Point **each_point**

Description: Iterates over the points that make up the simple polygon

ellipse

Signature: *[static]* new SimplePolygon ptr **ellipse** (const Box box, int n)

Description: Creates a simple polygon approximating an ellipse

box: The bounding box of the ellipse

n: The number of points that will be used to approximate the ellipse

This method has been introduced in version 0.23.

extract_rad

Signature: *[const]* variant[] **extract_rad**

Description: Extracts the corner radii from a rounded polygon

Attempts to extract the radii of rounded corner polygon. This is essentially the inverse of the round_corners method. If this method succeeds, if will return an array of four elements:

- The polygon with the rounded corners replaced by edgy ones
- The radius of the inner corners
- The radius of the outer corners
- The number of points per full circle

This method is based on some assumptions and may fail. In this case, an empty array is returned.

If successful, the following code will more or less render the original polygon and parameters

```
p = ...   # some polygon
p.round_corners(ri, ro, n)
(p2, ri2, ro2, n2) = p.extract_rad
# -> p2 == p, ro2 == ro, ri2 == ri, n2 == n (within some limits)
```

This method was introduced in version 0.25.

from_dpoly

Signature: *[static]* new SimplePolygon ptr **from_dpoly** (const DSimplePolygon dpolygon)

Description: Creates an integer coordinate polygon from a floating-point coordinate polygon

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dpoly'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new SimplePolygon ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given polygon. This method enables polygons as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

inside?

Signature: *[const]* bool **inside?** (Point p)

Description: Gets a value indicating whether the given point is inside the polygon

If the given point is inside or on the edge the polygon, true is returned. This tests works well only if the polygon is not self-overlapping and oriented clockwise.

is_box?

Signature: *[const]* bool **is_box?**

Description: Returns a value indicating whether the polygon is a simple box.

Returns: True if the polygon is a box.

A polygon is a box if it is identical to its bounding box.

This method was introduced in version 0.23.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_empty?

Signature: *[const]* bool **is_empty?**

Description: Returns a value indicating whether the polygon is empty

is_halfmanhattan?

Signature: *[const]* bool **is_halfmanhattan?**

Description: Returns a value indicating whether the polygon is half-manhattan

Half-manhattan polygons have edges which are multiples of 45 degree. These polygons can be clipped at a rectangle without potential grid snapping.

This predicate was introduced in version 0.27.

is_rectilinear?

Signature: *[const]* bool **is_rectilinear?**

Description: Returns a value indicating whether the polygon is rectilinear

minkowski_sum

(1) Signature: *[const]* Polygon **minkowski_sum** (const Edge e, bool resolve_holes)

Description: Computes the Minkowski sum of a polygon and an edge

e:	The edge.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and e.

This method was introduced in version 0.22.

(2) Signature: *[const]* Polygon **minkowski_sum** (const SimplePolygon p, bool resolve_holes)

Description: Computes the Minkowski sum of a polygon and a polygon

p:	The other polygon.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and p.

This method was introduced in version 0.22.

(3) Signature: *[const]* Polygon **minkowski_sum** (const Box b, bool resolve_holes)

Description: Computes the Minkowski sum of a polygon and a box

b:	The box.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and b.

This method was introduced in version 0.22.

(4) Signature: *[const]* Polygon **minkowski_sum** (Point[] c, bool resolve_holes)

Description: Computes the Minkowski sum of a polygon and a contour of points (a trace)

c:	The contour (a series of points forming the trace).
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and c.

This method was introduced in version 0.22.

minkowsky_sum

(1) Signature: *[const]* Polygon **minkowsky_sum** (const Edge e, bool resolve_holes)

Description: Computes the Minkowski sum of a polygon and an edge

e:	The edge.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and e.

Use of this method is deprecated. Use minkowski_sum instead

This method was introduced in version 0.22.

(2) Signature: *[const]* Polygon **minkowsky_sum** (const SimplePolygon p, bool resolve_holes)

Description: Computes the Minkowski sum of a polygon and a polygon

p:	The other polygon.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and p.

Use of this method is deprecated. Use minkowski_sum instead

This method was introduced in version 0.22.

(3) Signature: *[const]* Polygon **minkowsky_sum** (const Box b, bool resolve_holes)

Description: Computes the Minkowski sum of a polygon and a box

b:	The box.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and b.

Use of this method is deprecated. Use minkowski_sum instead

This method was introduced in version 0.22.

(4) Signature: *[const]* Polygon **minkowsky_sum** (Point[] c, bool resolve_holes)

Description: Computes the Minkowski sum of a polygon and a contour of points (a trace)

c:	The contour (a series of points forming the trace).
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and c.

Use of this method is deprecated. Use minkowski_sum instead

This method was introduced in version 0.22.

move

(1) Signature: SimplePolygon **move** (const Vector p)

Description: Moves the simple polygon.

p: The distance to move the simple polygon.

Returns: The moved simple polygon.

Moves the simple polygon by the given offset and returns the moved simple polygon. The polygon is overwritten.

(2) Signature: SimplePolygon **move** (int x, int y)

Description: Moves the polygon.

x: The x distance to move the polygon.

y: The y distance to move the polygon.

Returns: The moved polygon (self).

Moves the polygon by the given offset and returns the moved polygon. The polygon is overwritten.

moved

(1) Signature: *[const]* SimplePolygon **moved** (const Vector p)

Description: Returns the moved simple polygon

p: The distance to move the simple polygon.

Returns: The moved simple polygon.

Moves the simple polygon by the given offset and returns the moved simple polygon. The polygon is not modified.

(2) Signature: *[const]* SimplePolygon **moved** (int x, int y)

Description: Returns the moved polygon (does not modify self)

x: The x distance to move the polygon.

y: The y distance to move the polygon.

Returns: The moved polygon.

Moves the polygon by the given offset and returns the moved polygon. The polygon is not modified.

This method has been introduced in version 0.23.

new

(1) Signature: *[static]* new SimplePolygon ptr **new** (const DSimplePolygon dpolygon)

Description: Creates an integer coordinate polygon from a floating-point coordinate polygon

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dpoly'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new SimplePolygon ptr **new**

Description: Default constructor: creates an empty (invalid) polygon

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new SimplePolygon ptr **new** (Point[] pts, bool raw = false)

Description: Constructor given the points of the simple polygon

pts: The points forming the simple polygon

raw: If true, the points are taken as they are (see below)

If the 'raw' argument is set to true, the points are taken as they are. Specifically no removal of redundant points or joining of coincident edges will take place. In effect, polygons consisting of a single point or two points can be constructed as well as polygons with duplicate points. Note that such polygons may cause problems in some applications.

Regardless of raw mode, the point list will be adjusted such that the first point is the lowest-leftmost one and the orientation is clockwise always.

The 'raw' argument has been added in version 0.24.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new SimplePolygon ptr **new** (const Box box)

Description: Constructor converting a box to a polygon

box: The box to convert to a polygon

Python specific notes:
This method is the default initializer of the object.

num_points

Signature: *[const]* unsigned long **num_points**

Description: Gets the number of points

perimeter

Signature: *[const]* unsigned long **perimeter**

Description: Gets the perimeter of the polygon

The perimeter is sum of the lengths of all edges making up the polygon.

point

Signature: *[const]* Point **point** (unsigned long p)

Description: Gets a specific point of the contour@param p The index of the point to get

If the index of the point is not a valid index, a default value is returned. This method was introduced in version 0.18.

points=

Signature: void **points=** (Point[] pts)

Description: Sets the points of the simple polygon

pts: An array of points to assign to the simple polygon

See the constructor description for details about raw mode.

Python specific notes:
The object exposes a writable attribute 'points'. This is the setter.

round_corners

Signature: *[const]* SimplePolygon **round_corners** (double rinner, double router, unsigned int n)

Description: Rounds the corners of the polygon

rinner: The circle radius of inner corners (in database units).

router: The circle radius of outer corners (in database units).

n: The number of points per full circle.

Returns: The new polygon.

Replaces the corners of the polygon with circle segments.

This method was introduced in version 0.22 for integer coordinates and in 0.25 for all coordinate types.

set_points

Signature: void **set_points** (Point[] pts, bool raw = false)

Description: Sets the points of the simple polygon

pts: An array of points to assign to the simple polygon

raw: If true, the points are taken as they are

See the constructor description for details about raw mode.

This method has been added in version 0.24.

split

Signature: *[const]* SimplePolygon[] **split**

Description: Splits the polygon into two or more parts

This method will break the polygon into parts. The exact breaking algorithm is unspecified, the result are smaller polygons of roughly equal number of points and 'less concave' nature. Usually the returned polygon set consists of two polygons, but there can be more. The merged region of the resulting polygons equals the original polygon with the exception of small snapping effects at new vertexes.

The intended use for this method is a iteratively split polygons until the satisfy some maximum number of points limit.

This method has been introduced in version 0.25.3.

to_dtype

Signature: *[const]* DSimplePolygon **to_dtype** (double dbu = 1)

Description: Converts the polygon to a floating-point coordinate polygon

The database unit can be specified to translate the integer-coordinate polygon into a floating-point coordinate polygon in micron units. The database unit is basically a scaling factor.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s**

Description: Returns a string representing the polygon

Python specific notes:
This method is also available as 'str(object)'.

touches?

(1) Signature: *[const]* bool **touches?** (const Box box)

Description: Returns true, if the polygon touches the given box.

The box and the polygon touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

(2) Signature: *[const]* bool **touches?** (const Edge edge)

Description: Returns true, if the polygon touches the given edge.

The edge and the polygon touch if they overlap or the edge shares at least one point with the polygon's contour.

This method was introduced in version 0.25.1.

(3) Signature: *[const]* bool **touches?** (const Polygon polygon)

Description: Returns true, if the polygon touches the other polygon.

The polygons touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

(4) Signature: *[const]* bool **touches?** (const SimplePolygon simple_polygon)

Description: Returns true, if the polygon touches the other polygon.

The polygons touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

transform

(1) Signature: SimplePolygon ptr **transform** (const ICplxTrans t)

Description: Transforms the simple polygon with a complex transformation (in-place)

t: The transformation to apply.

Transforms the simple polygon with the given complex transformation. Modifies self and returns self. An out-of-place version which does not modify self is transformed.

This method has been introduced in version 0.24.

(2) Signature: SimplePolygon ptr **transform** (const Trans t)

Description: Transforms the simple polygon (in-place)

t: The transformation to apply.

Transforms the simple polygon with the given transformation. Modifies self and returns self. An out-of-place version which does not modify self is transformed.

This method has been introduced in version 0.24.

transformed

(1) Signature: *[const]* SimplePolygon **transformed** (const ICplxTrans t)

Description: Transforms the simple polygon.

t: The transformation to apply.

Returns: The transformed simple polygon (in this case an integer coordinate object).

Transforms the simple polygon with the given complex transformation. Does not modify the simple polygon but returns the transformed polygon.

This method has been introduced in version 0.18.

(2) Signature: *[const]* SimplePolygon **transformed** (const Trans t)

Description: Transforms the simple polygon.

t: The transformation to apply.

Returns: The transformed simple polygon.

Transforms the simple polygon with the given transformation. Does not modify the simple polygon but returns the transformed polygon.

(3) Signature: *[const]* DSimplePolygon **transformed** (const CplxTrans t)

Description: Transforms the simple polygon.

t: The transformation to apply.

Returns: The transformed simple polygon.

Transforms the simple polygon with the given complex transformation. Does not modify the simple polygon but returns the transformed polygon.

With version 0.25, the original 'transformed_cplx' method is deprecated and 'transformed' takes both simple and complex transformations.

transformed_cplx

Signature: *[const]* DSimplePolygon **transformed_cplx** (const CplxTrans t)

Description: Transforms the simple polygon.

t:	The transformation to apply.
Returns:	The transformed simple polygon.

Use of this method is deprecated. Use transformed instead

Transforms the simple polygon with the given complex transformation. Does not modify the simple polygon but returns the transformed polygon.

With version 0.25, the original 'transformed_cplx' method is deprecated and 'transformed' takes both simple and complex transformations.

4.75. API reference - Class DSimplePolygon

Notation used in Ruby API documentation

Module: db

Description: A simple polygon class

A simple polygon consists of an outer hull only. To support polygons with holes, use DPolygon. The contour consists of several points. The point list is normalized such that the leftmost, lowest point is the first one. The orientation is normalized such that the orientation of the hull contour is clockwise.

It is in no way checked that the contours are not over- lapping. This must be ensured by the user of the object when filling the contours.

The DSimplePolygon class stores coordinates in floating-point format which gives a higher precision for some operations. A class that stores integer coordinates is SimplePolygon.

See The Database API for more details about the database objects.

Public constructors

new DSimplePolygon ptr	**new**	(const SimplePolygon polygon)	Creates a floating-point coordinate polygon from an integer coordinate polygon
new DSimplePolygon ptr	**new**		Default constructor: creates an empty (invalid) polygon
new DSimplePolygon ptr	**new**	(DPoint[] pts, bool raw = false)	Constructor given the points of the simple polygon
new DSimplePolygon ptr	**new**	(const DBox box)	Constructor converting a box to a polygon

Public methods

[const]	bool	**!=**	(const DSimplePolygon p)	Returns a value indicating whether self is not equal to p
[const]	DSimplePolygon	*****	(double f)	Scales the polygon by some factor
[const]	bool	**<**	(const DSimplePolygon p)	Returns a value indicating whether self is less than p
[const]	bool	**==**	(const DSimplePolygon p)	Returns a value indicating whether self is equal to p
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.

	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	double	area		Gets the area of the polygon
[const]	double	area2		Gets the double area of the polygon
	void	assign	(const DSimplePolygon other)	Assigns another object to self
[const]	DBox	bbox		Returns the bounding box of the simple polygon
[const]	DSimplePolygon[]	break	(unsigned long max_vertex_count, double max_area_ratio)	Splits the polygon into parts with a maximum vertex count and area ratio
	void	compress	(bool remove_reflected)	Compressed the simple polygon.
[const]	new DSimplePolygon ptr	dup		Creates a copy of self
[const,iter]	DEdge	each_edge		Iterates over the edges that make up the simple polygon
[const,iter]	DPoint	each_point		Iterates over the points that make up the simple polygon
[const]	variant[]	extract_rad		Extracts the corner radii from a rounded polygon
[const]	unsigned long	hash		Computes a hash value
[const]	bool	inside?	(DPoint p)	Gets a value indicating whether the given point is inside the polygon
[const]	bool	is_box?		Returns a value indicating whether the polygon is a simple box.
[const]	bool	is_empty?		Returns a value indicating whether the polygon is empty
[const]	bool	is_halfmanhattan?		Returns a value indicating whether the polygon is half-manhattan
[const]	bool	is_rectilinear?		Returns a value indicating whether the polygon is rectilinear
	DSimplePolygon	move	(const DVector p)	Moves the simple polygon.
	DSimplePolygon	move	(double x, double y)	Moves the polygon.
[const]	DSimplePolygon	moved	(const DVector p)	Returns the moved simple polygon

[const]	DSimplePolygon	**moved**	(double x, double y)	Returns the moved polygon (does not modify self)
[const]	unsigned long	**num_points**		Gets the number of points
[const]	double	**perimeter**		Gets the perimeter of the polygon
[const]	DPoint	**point**	(unsigned long p)	Gets a specific point of the contour@param p The index of the point to get
	void	**points=**	(DPoint[] pts)	Sets the points of the simple polygon
[const]	DSimplePolygon	**round_corners**	(double rinner, double router, unsigned int n)	Rounds the corners of the polygon
	void	**set_points**	(DPoint[] pts, bool raw = false)	Sets the points of the simple polygon
[const]	DSimplePolygon[]	**split**		Splits the polygon into two or more parts
[const]	SimplePolygon	**to_itype**	(double dbu = 1)	Converts the polygon to an integer coordinate polygon
[const]	string	**to_s**		Returns a string representing the polygon
[const]	bool	**touches?**	(const DBox box)	Returns true, if the polygon touches the given box.
[const]	bool	**touches?**	(const DEdge edge)	Returns true, if the polygon touches the given edge.
[const]	bool	**touches?**	(const DPolygon polygon)	Returns true, if the polygon touches the other polygon.
[const]	bool	**touches?**	(const DSimplePolygon simple_polygon)	Returns true, if the polygon touches the other polygon.
	DSimplePolygon ptr	**transform**	(const DCplxTrans t)	Transforms the simple polygon with a complex transformation (in-place)
	DSimplePolygon ptr	**transform**	(const DTrans t)	Transforms the simple polygon (in-place)
[const]	SimplePolygon	**transformed**	(const VCplxTrans t)	Transforms the polygon with the given complex transformation
[const]	DSimplePolygon	**transformed**	(const DTrans t)	Transforms the simple polygon.
[const]	DSimplePolygon	**transformed**	(const DCplxTrans t)	Transforms the simple polygon.

Public static methods and constants

new DSimplePolygon ptr	**ellipse**	(const DBox box, int n)	Creates a simple polygon approximating an ellipse

	new DSimplePolygon ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new DSimplePolygon ptr	**from_ipoly**	(const SimplePolygon polygon)	Use of this method is deprecated. Use new instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	DSimplePolygon	**transformed_cplx**	(const DCplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

!=

Signature: *[const]* bool **!=** (const DSimplePolygon p)

Description: Returns a value indicating whether self is not equal to p

p: The object to compare against

*****

Signature: *[const]* DSimplePolygon ***** (double f)

Description: Scales the polygon by some factor

Returns the scaled object. All coordinates are multiplied with the given factor and if necessary rounded.

Python specific notes:
This method also implements '__rmul__'.

<

Signature: *[const]* bool **<** (const DSimplePolygon p)

Description: Returns a value indicating whether self is less than p

p: The object to compare against

This operator is provided to establish some, not necessarily a certain sorting order

This method has been introduced in version 0.25.

==

Signature: *[const]* bool **==** (const DSimplePolygon p)

Description: Returns a value indicating whether self is equal to p

p: The object to compare against

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

area

Signature: *[const]* double **area**

Description: Gets the area of the polygon

The area is correct only if the polygon is not self-overlapping and the polygon is oriented clockwise.

area2

Signature: *[const]* double **area2**

Description: Gets the double area of the polygon

This method is provided because the area for an integer-type polygon is a multiple of 1/2. Hence the double area can be expresses precisely as an integer for these types.

This method has been introduced in version 0.26.1

assign

Signature: void **assign** (const DSimplePolygon other)

Description: Assigns another object to self

bbox

Signature: *[const]* DBox **bbox**

Description: Returns the bounding box of the simple polygon

break

Signature: *[const]* DSimplePolygon[] **break** (unsigned long max_vertex_count, double max_area_ratio)

Description: Splits the polygon into parts with a maximum vertex count and area ratio

The area ratio is the ratio between the bounding box area and the polygon area. Higher values mean more 'skinny' polygons.

This method will split the input polygon into pieces having a maximum of 'max_vertex_count' vertices and an area ratio less than 'max_area_ratio'. 'max_vertex_count' can be zero. In this case the limit is ignored. Also 'max_area_ratio' can be zero, in which case it is ignored as well.

The method of splitting is unspecified. The algorithm will apply 'split' recursively until the parts satisfy the limits.

This method has been introduced in version 0.29.

Python specific notes:
This attribute is available as 'break_' in Python.

compress

Signature: void **compress** (bool remove_reflected)

Description: Compressed the simple polygon.

remove_reflected: See description of the functionality.

This method removes redundant points from the polygon, such as points being on a line formed by two other points. If remove_reflected is true, points are also removed if the two adjacent edges form a spike.

This method was introduced in version 0.18.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new DSimplePolygon ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_edge

Signature: *[const,iter]* DEdge **each_edge**

Description: Iterates over the edges that make up the simple polygon

each_point

Signature: *[const,iter]* DPoint **each_point**

Description: Iterates over the points that make up the simple polygon

ellipse

Signature: *[static]* new DSimplePolygon ptr **ellipse** (const DBox box, int n)

Description: Creates a simple polygon approximating an ellipse

box: The bounding box of the ellipse

n: The number of points that will be used to approximate the ellipse

This method has been introduced in version 0.23.

extract_rad

Signature: *[const]* variant[] **extract_rad**

Description: Extracts the corner radii from a rounded polygon

Attempts to extract the radii of rounded corner polygon. This is essentially the inverse of the round_corners method. If this method succeeds, if will return an array of four elements:

- The polygon with the rounded corners replaced by edgy ones
- The radius of the inner corners
- The radius of the outer corners
- The number of points per full circle

This method is based on some assumptions and may fail. In this case, an empty array is returned.

If successful, the following code will more or less render the original polygon and parameters

```
p = ...   # some polygon
p.round_corners(ri, ro, n)
(p2, ri2, ro2, n2) = p.extract_rad
# -> p2 == p, ro2 == ro, ri2 == ri, n2 == n (within some limits)
```

This method was introduced in version 0.25.

from_ipoly

Signature: *[static]* new DSimplePolygon ptr **from_ipoly** (const SimplePolygon polygon)

Description: Creates a floating-point coordinate polygon from an integer coordinate polygon

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_ipoly'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new DSimplePolygon ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given polygon. This method enables polygons as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

inside?

Signature: *[const]* bool **inside?** (DPoint p)

Description: Gets a value indicating whether the given point is inside the polygon

If the given point is inside or on the edge the polygon, true is returned. This tests works well only if the polygon is not self-overlapping and oriented clockwise.

is_box?

Signature: *[const]* bool **is_box?**

Description: Returns a value indicating whether the polygon is a simple box.

Returns: True if the polygon is a box.

A polygon is a box if it is identical to its bounding box.

This method was introduced in version 0.23.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_empty?

Signature: *[const]* bool **is_empty?**

Description: Returns a value indicating whether the polygon is empty

is_halfmanhattan?

Signature: *[const]* bool **is_halfmanhattan?**

Description: Returns a value indicating whether the polygon is half-manhattan

Half-manhattan polygons have edges which are multiples of 45 degree. These polygons can be clipped at a rectangle without potential grid snapping.

This predicate was introduced in version 0.27.

is_rectilinear?

Signature: *[const]* bool **is_rectilinear?**

Description: Returns a value indicating whether the polygon is rectilinear

move

(1) Signature: DSimplePolygon **move** (const DVector p)

Description: Moves the simple polygon.

p: The distance to move the simple polygon.

Returns: The moved simple polygon.

Moves the simple polygon by the given offset and returns the moved simple polygon. The polygon is overwritten.

(2) Signature: DSimplePolygon **move** (double x, double y)

Description: Moves the polygon.

x: The x distance to move the polygon.

y: The y distance to move the polygon.

Returns: The moved polygon (self).

Moves the polygon by the given offset and returns the moved polygon. The polygon is overwritten.

moved

(1) Signature: *[const]* DSimplePolygon **moved** (const DVector p)

Description: Returns the moved simple polygon

p: The distance to move the simple polygon.

Returns: The moved simple polygon.

Moves the simple polygon by the given offset and returns the moved simple polygon. The polygon is not modified.

(2) Signature: *[const]* DSimplePolygon **moved** (double x, double y)

Description: Returns the moved polygon (does not modify self)

x: The x distance to move the polygon.

y: The y distance to move the polygon.

Returns: The moved polygon.

Moves the polygon by the given offset and returns the moved polygon. The polygon is not modified.

This method has been introduced in version 0.23.

new

(1) Signature: *[static]* new DSimplePolygon ptr **new** (const SimplePolygon polygon)

Description: Creates a floating-point coordinate polygon from an integer coordinate polygon

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_ipoly'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DSimplePolygon ptr **new**

Description: Default constructor: creates an empty (invalid) polygon

Python specific notes:

This method is the default initializer of the object.

(3) Signature: *[static]* new DSimplePolygon ptr **new** (DPoint[] pts, bool raw = false)

Description: Constructor given the points of the simple polygon

pts:	The points forming the simple polygon
raw:	If true, the points are taken as they are (see below)

If the 'raw' argument is set to true, the points are taken as they are. Specifically no removal of redundant points or joining of coincident edges will take place. In effect, polygons consisting of a single point or two points can be constructed as well as polygons with duplicate points. Note that such polygons may cause problems in some applications.

Regardless of raw mode, the point list will be adjusted such that the first point is the lowest-leftmost one and the orientation is clockwise always.

The 'raw' argument has been added in version 0.24.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new DSimplePolygon ptr **new** (const DBox box)

Description: Constructor converting a box to a polygon

box:	The box to convert to a polygon

Python specific notes:
This method is the default initializer of the object.

num_points

Signature: *[const]* unsigned long **num_points**

Description: Gets the number of points

perimeter

Signature: *[const]* double **perimeter**

Description: Gets the perimeter of the polygon

The perimeter is sum of the lengths of all edges making up the polygon.

point

Signature: *[const]* DPoint **point** (unsigned long p)

Description: Gets a specific point of the contour@param p The index of the point to get

If the index of the point is not a valid index, a default value is returned. This method was introduced in version 0.18.

points=

Signature: void **points=** (DPoint[] pts)

Description: Sets the points of the simple polygon

pts:	An array of points to assign to the simple polygon

See the constructor description for details about raw mode.

Python specific notes:
The object exposes a writable attribute 'points'. This is the setter.

round_corners

Signature: *[const]* DSimplePolygon **round_corners** (double rinner, double router, unsigned int n)

Description: Rounds the corners of the polygon

rinner:	The circle radius of inner corners (in database units).
router:	The circle radius of outer corners (in database units).

n: The number of points per full circle.

Returns: The new polygon.

Replaces the corners of the polygon with circle segments.

This method was introduced in version 0.22 for integer coordinates and in 0.25 for all coordinate types.

set_points

Signature: void **set_points** (DPoint[] pts, bool raw = false)

Description: Sets the points of the simple polygon

pts: An array of points to assign to the simple polygon

raw: If true, the points are taken as they are

See the constructor description for details about raw mode.

This method has been added in version 0.24.

split

Signature: *[const]* DSimplePolygon[] **split**

Description: Splits the polygon into two or more parts

This method will break the polygon into parts. The exact breaking algorithm is unspecified, the result are smaller polygons of roughly equal number of points and 'less concave' nature. Usually the returned polygon set consists of two polygons, but there can be more. The merged region of the resulting polygons equals the original polygon with the exception of small snapping effects at new vertexes.

The intended use for this method is a iteratively split polygons until the satisfy some maximum number of points limit.

This method has been introduced in version 0.25.3.

to_itype

Signature: *[const]* SimplePolygon **to_itype** (double dbu = 1)

Description: Converts the polygon to an integer coordinate polygon

The database unit can be specified to translate the floating-point coordinate polygon in micron units to an integer-coordinate polygon in database units. The polygon's' coordinates will be divided by the database unit.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s**

Description: Returns a string representing the polygon

Python specific notes:
This method is also available as 'str(object)'.

touches?

(1) Signature: *[const]* bool **touches?** (const DBox box)

Description: Returns true, if the polygon touches the given box.

The box and the polygon touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

(2) Signature: *[const]* bool **touches?** (const DEdge edge)

Description: Returns true, if the polygon touches the given edge.

The edge and the polygon touch if they overlap or the edge shares at least one point with the polygon's contour.

This method was introduced in version 0.25.1.

(3) Signature: *[const]* bool **touches?** (const DPolygon polygon)

Description: Returns true, if the polygon touches the other polygon.

The polygons touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

(4) Signature: *[const]* bool **touches?** (const DSimplePolygon simple_polygon)

Description: Returns true, if the polygon touches the other polygon.

The polygons touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

transform

(1) Signature: DSimplePolygon ptr **transform** (const DCplxTrans t)

Description: Transforms the simple polygon with a complex transformation (in-place)

t: The transformation to apply.

Transforms the simple polygon with the given complex transformation. Modifies self and returns self. An out-of-place version which does not modify self is transformed.

This method has been introduced in version 0.24.

(2) Signature: DSimplePolygon ptr **transform** (const DTrans t)

Description: Transforms the simple polygon (in-place)

t: The transformation to apply.

Transforms the simple polygon with the given transformation. Modifies self and returns self. An out-of-place version which does not modify self is transformed.

This method has been introduced in version 0.24.

transformed

(1) Signature: *[const]* SimplePolygon **transformed** (const VCplxTrans t)

Description: Transforms the polygon with the given complex transformation

t: The magnifying transformation to apply

Returns: The transformed polygon (in this case an integer coordinate polygon)

This method has been introduced in version 0.25.

(2) Signature: *[const]* DSimplePolygon **transformed** (const DTrans t)

Description: Transforms the simple polygon.

t: The transformation to apply.

Returns: The transformed simple polygon.

Transforms the simple polygon with the given transformation. Does not modify the simple polygon but returns the transformed polygon.

(3) Signature: *[const]* DSimplePolygon **transformed** (const DCplxTrans t)

Description: Transforms the simple polygon.

t: The transformation to apply.

Returns: The transformed simple polygon.

Transforms the simple polygon with the given complex transformation. Does not modify the simple polygon but returns the transformed polygon.

With version 0.25, the original 'transformed_cplx' method is deprecated and 'transformed' takes both simple and complex transformations.

transformed_cplx

Signature: *[const]* DSimplePolygon **transformed_cplx** (const DCplxTrans t)

Description: Transforms the simple polygon.

t:	The transformation to apply.
Returns:	The transformed simple polygon.

Use of this method is deprecated. Use transformed instead

Transforms the simple polygon with the given complex transformation. Does not modify the simple polygon but returns the transformed polygon.

With version 0.25, the original 'transformed_cplx' method is deprecated and 'transformed' takes both simple and complex transformations.

4.76. API reference - Class Polygon

Notation used in Ruby API documentation

Module: db

Description: A polygon class

A polygon consists of an outer hull and zero to many holes. Each contour consists of several points. The point list is normalized such that the leftmost, lowest point is the first one. The orientation is normalized such that the orientation of the hull contour is clockwise, while the orientation of the holes is counterclockwise.

It is in no way checked that the contours are not overlapping. This must be ensured by the user of the object when filling the contours.

A polygon can be asked for the number of holes using the holes method. each_point_hull delivers the points of the hull contour. each_point_hole delivers the points of a specific hole. each_edge delivers the edges (point-to-point connections) of both hull and holes. bbox delivers the bounding box, area the area and perimeter the perimeter of the polygon.

Here's an example of how to create a polygon:

```
hull =  [ RBA::Point::new(0, 0),       RBA::Point::new(6000, 0),
          RBA::Point::new(6000, 3000), RBA::Point::new(0, 3000) ]
hole1 = [ RBA::Point::new(1000, 1000), RBA::Point::new(2000, 1000),
          RBA::Point::new(2000, 2000), RBA::Point::new(1000, 2000) ]
hole2 = [ RBA::Point::new(3000, 1000), RBA::Point::new(4000, 1000),
          RBA::Point::new(4000, 2000), RBA::Point::new(3000, 2000) ]
poly = RBA::Polygon::new(hull)
poly.insert_hole(hole1)
poly.insert_hole(hole2)

# ask the polygon for some properties
poly.holes      # -> 2
poly.area       # -> 16000000
poly.perimeter  # -> 26000
poly.bbox       # -> (0,0;6000,3000)
```

The Polygon class stores coordinates in integer format. A class that stores floating-point coordinates is DPolygon.

See The Database API for more details about the database objects.

Public constructors

new Polygon ptr	**new**	(const DPolygon dpolygon)	Creates an integer coordinate polygon from a floating-point coordinate polygon
new Polygon ptr	**new**		Creates an empty (invalid) polygon
new Polygon ptr	**new**	(const SimplePolygon sp)	Creates a polygon from a simple polygon
new Polygon ptr	**new**	(Point[] pts, bool raw = false)	Creates a polygon from a point array for the hull
new Polygon ptr	**new**	(const Box box)	Creates a polygon from a box

Public methods

[const]	bool	**!=**	(const Polygon p)	Returns a value indicating whether the polygons are not equal
[const]	Polygon	*****	(double f)	Scales the polygon by some factor

[const]	bool	**<**	(const Polygon p)	Returns a value indicating whether self is less than p
[const]	bool	**==**	(const Polygon p)	Returns a value indicating whether the polygons are equal
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	long	**area**		Gets the area of the polygon
[const]	long	**area2**		Gets the double area of the polygon
	void	**assign**	(const Polygon other)	Assigns another object to self
	void	**assign_hole**	(unsigned int n, Point[] p, bool raw = false)	Sets the points of the given hole of the polygon
	void	**assign_hole**	(unsigned int n, const Box b)	Sets the box as the given hole of the polygon
	void	**assign_hull**	(Point[] p, bool raw = false)	Sets the points of the hull of polygon
[const]	Box	**bbox**		Returns the bounding box of the polygon
[const]	Polygon[]	**break**	(unsigned long max_vertex_count, double max_area_ratio)	Splits the polygon into parts with a maximum vertex count and area ratio
	void	**compress**	(bool remove_reflected)	Compresses the polygon.
[const]	SimplePolygon[]	**decompose_convex**	(int preferred_orientation = PO_any)	Decomposes the polygon into convex pieces
[const]	SimplePolygon[]	**decompose_trapezoi**	(int mode = TD_simple)	Decomposes the polygon into trapezoids
[const]	new Polygon ptr	**dup**		Creates a copy of self
[const,iter]	Edge	**each_edge**		Iterates over the edges that make up the polygon

[const,iter]	Edge	**each_edge**	(unsigned int contour)	Iterates over the edges of one contour of the polygon
[const,iter]	Point	**each_point_hole**	(unsigned int n)	Iterates over the points that make up the nth hole
[const,iter]	Point	**each_point_hull**		Iterates over the points that make up the hull
[const]	variant[]	**extract_rad**		Extracts the corner radii from a rounded polygon
[const]	unsigned long	**hash**		Computes a hash value
[const]	unsigned int	**holes**		Returns the number of holes
	void	**hull=**	(Point[] p)	Sets the points of the hull of polygon
	void	**insert_hole**	(Point[] p, bool raw = false)	Inserts a hole with the given points
	void	**insert_hole**	(const Box b)	Inserts a hole from the given box
[const]	bool	**inside?**	(Point p)	Tests, if the given point is inside the polygon
[const]	bool	**is_box?**		Returns true, if the polygon is a simple box.
[const]	bool	**is_convex?**		Returns a value indicating whether the polygon is convex
[const]	bool	**is_empty?**		Returns a value indicating whether the polygon is empty
[const]	bool	**is_halfmanhattan?**		Returns a value indicating whether the polygon is half-manhattan
[const]	bool	**is_rectilinear?**		Returns a value indicating whether the polygon is rectilinear
[const]	Polygon	**minkowski_sum**	(const Edge e, bool resolve_holes)	Computes the Minkowski sum of the polygon and an edge
[const]	Polygon	**minkowski_sum**	(const Polygon b, bool resolve_holes)	Computes the Minkowski sum of the polygon and a polygon
[const]	Polygon	**minkowski_sum**	(const Box b, bool resolve_holes)	Computes the Minkowski sum of the polygon and a box
[const]	Polygon	**minkowski_sum**	(Point[] b, bool resolve_holes)	Computes the Minkowski sum of the polygon and a contour of points (a trace)
	Polygon	**move**	(const Vector p)	Moves the polygon.
	Polygon	**move**	(int x, int y)	Moves the polygon.

[const]	Polygon	**moved**	(const Vector p)	Returns the moved polygon (does not modify self)
[const]	Polygon	**moved**	(int x, int y)	Returns the moved polygon (does not modify self)
[const]	unsigned long	**num_points**		Gets the total number of points (hull plus holes)
[const]	unsigned long	**num_points_hole**	(unsigned int n)	Gets the number of points of the given hole
[const]	unsigned long	**num_points_hull**		Gets the number of points of the hull
[const]	unsigned long	**perimeter**		Gets the perimeter of the polygon
[const]	Point	**point_hole**	(unsigned int n, unsigned long p)	Gets a specific point of a hole
[const]	Point	**point_hull**	(unsigned long p)	Gets a specific point of the hull
	void	**resolve_holes**		Resolve holes by inserting cut lines and joining the holes with the hull
[const]	Polygon	**resolved_holes**		Returns a polygon without holes
[const]	Polygon	**round_corners**	(double rinner, double router, unsigned int n)	Rounds the corners of the polygon
	void	**size**	(int dx, int dy, unsigned int mode)	Sizes the polygon (biasing)
	void	**size**	(const Vector dv, unsigned int mode = 2)	Sizes the polygon (biasing)
	void	**size**	(int d, unsigned int mode = 2)	Sizes the polygon (biasing)
[const]	Polygon	**sized**	(int dx, int dy, unsigned int mode)	Sizes the polygon (biasing) without modifying self
[const]	Polygon	**sized**	(const Vector dv, unsigned int mode = 2)	Sizes the polygon (biasing) without modifying self
[const]	Polygon	**sized**	(int d, unsigned int mode = 2)	Sizes the polygon (biasing) without modifying self
[const]	Polygon	**smooth**	(int d, bool keep_hv = false)	Smooths a polygon
	void	**sort_holes**		Brings the holes in a specific order

[const]	Polygon[]	**split**		Splits the polygon into two or more parts
[const]	DPolygon	**to_dtype**	(double dbu = 1)	Converts the polygon to a floating-point coordinate polygon
[const]	string	**to_s**		Returns a string representing the polygon
[const]	SimplePolygon	**to_simple_polygon**		Converts a polygon to a simple polygon
[const]	bool	**touches?**	(const Box box)	Returns true, if the polygon touches the given box.
[const]	bool	**touches?**	(const Edge edge)	Returns true, if the polygon touches the given edge.
[const]	bool	**touches?**	(const Polygon polygon)	Returns true, if the polygon touches the other polygon.
[const]	bool	**touches?**	(const SimplePolygon simple_polygon)	Returns true, if the polygon touches the other polygon.
	Polygon ptr	**transform**	(const ICplxTrans t)	Transforms the polygon with a complex transformation (in-place)
	Polygon ptr	**transform**	(const Trans t)	Transforms the polygon (in-place)
[const]	Polygon	**transformed**	(const Trans t)	Transforms the polygon
[const]	DPolygon	**transformed**	(const CplxTrans t)	Transforms the polygon with a complex transformation

Public static methods and constants

[static,const]	int	**PO_any**	A value for the preferred orientation parameter of decompose_convex
[static,const]	int	**PO_horizontal**	A value for the preferred orientation parameter of decompose_convex
[static,const]	int	**PO_htrapezoids**	A value for the preferred orientation parameter of decompose_convex
[static,const]	int	**PO_vertical**	A value for the preferred orientation parameter of decompose_convex
[static,const]	int	**PO_vtrapezoids**	A value for the preferred orientation parameter of decompose_convex
[static,const]	int	**TD_htrapezoids**	A value for the mode parameter of decompose_trapezoids
[static,const]	int	**TD_simple**	A value for the mode parameter of decompose_trapezoids
[static,const]	int	**TD_vtrapezoids**	A value for the mode parameter of decompose_trapezoids

	new Polygon ptr	**ellipse**	(const Box box, int n)	Creates a simple polygon approximating an ellipse
	new Polygon ptr	**from_s**	(string s)	Creates a polygon from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new Polygon ptr	**from_dpoly**	(const DPolygon dpolygon)	Use of this method is deprecated. Use new instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	Polygon	**minkowsky_sum**	(const Edge e, bool resolve_holes)	Use of this method is deprecated. Use minkowski_sum instead
[const]	Polygon	**minkowsky_sum**	(const Polygon b, bool resolve_holes)	Use of this method is deprecated. Use minkowski_sum instead
[const]	Polygon	**minkowsky_sum**	(const Box b, bool resolve_holes)	Use of this method is deprecated. Use minkowski_sum instead
[const]	Polygon	**minkowsky_sum**	(Point[] b, bool resolve_holes)	Use of this method is deprecated. Use minkowski_sum instead
[const]	Polygon	**transformed**	(const ICplxTrans t)	Use of this method is deprecated
[const]	DPolygon	**transformed_cplx**	(const CplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

!=

Signature: *[const]* bool **!=** (const Polygon p)

Description: Returns a value indicating whether the polygons are not equal

p: The object to compare against

*****

Signature: *[const]* Polygon ***** (double f)

Description: Scales the polygon by some factor

Returns the scaled object. All coordinates are multiplied with the given factor and if necessary rounded.

Python specific notes:

This method also implements '__rmul__'.

<	**Signature**: *[const]* bool **<** (const Polygon p) **Description**: Returns a value indicating whether self is less than p **p**: The object to compare against This operator is provided to establish some, not necessarily a certain sorting order
==	**Signature**: *[const]* bool **==** (const Polygon p) **Description**: Returns a value indicating whether the polygons are equal **p**: The object to compare against
PO_any	**Signature**: *[static,const]* int **PO_any** **Description**: A value for the preferred orientation parameter of decompose_convex This value indicates that there is not cut preference This constant has been introduced in version 0.25. **Python specific notes:** The object exposes a readable attribute 'PO_any'. This is the getter.
PO_horizontal	**Signature**: *[static,const]* int **PO_horizontal** **Description**: A value for the preferred orientation parameter of decompose_convex This value indicates that there only horizontal cuts are allowed This constant has been introduced in version 0.25. **Python specific notes:** The object exposes a readable attribute 'PO_horizontal'. This is the getter.
PO_htrapezoids	**Signature**: *[static,const]* int **PO_htrapezoids** **Description**: A value for the preferred orientation parameter of decompose_convex This value indicates that cuts shall favor decomposition into horizontal trapezoids This constant has been introduced in version 0.25. **Python specific notes:** The object exposes a readable attribute 'PO_htrapezoids'. This is the getter.
PO_vertical	**Signature**: *[static,const]* int **PO_vertical** **Description**: A value for the preferred orientation parameter of decompose_convex This value indicates that there only vertical cuts are allowed This constant has been introduced in version 0.25. **Python specific notes:** The object exposes a readable attribute 'PO_vertical'. This is the getter.
PO_vtrapezoids	**Signature**: *[static,const]* int **PO_vtrapezoids** **Description**: A value for the preferred orientation parameter of decompose_convex This value indicates that cuts shall favor decomposition into vertical trapezoids This constant has been introduced in version 0.25. **Python specific notes:** The object exposes a readable attribute 'PO_vtrapezoids'. This is the getter.

TD_htrapezoids

Signature: *[static,const]* int **TD_htrapezoids**

Description: A value for the mode parameter of decompose_trapezoids

This value indicates simple decomposition mode. This mode produces horizontal trapezoids and tries to minimize the number of trapezoids. This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'TD_htrapezoids'. This is the getter.

TD_simple

Signature: *[static,const]* int **TD_simple**

Description: A value for the mode parameter of decompose_trapezoids

This value indicates simple decomposition mode. This mode is fast but does not make any attempts to produce less trapezoids. This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'TD_simple'. This is the getter.

TD_vtrapezoids

Signature: *[static,const]* int **TD_vtrapezoids**

Description: A value for the mode parameter of decompose_trapezoids

This value indicates simple decomposition mode. This mode produces vertical trapezoids and tries to minimize the number of trapezoids.

Python specific notes:
The object exposes a readable attribute 'TD_vtrapezoids'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is

known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

area

Signature: *[const]* long **area**

Description: Gets the area of the polygon

The area is correct only if the polygon is not self-overlapping and the polygon is oriented clockwise.Orientation is ensured automatically in most cases.

area2

Signature: *[const]* long **area2**

Description: Gets the double area of the polygon

This method is provided because the area for an integer-type polygon is a multiple of 1/2. Hence the double area can be expresses precisely as an integer for these types.

This method has been introduced in version 0.26.1

assign

Signature: void **assign** (const Polygon other)

Description: Assigns another object to self

assign_hole

(1) Signature: void **assign_hole** (unsigned int n, Point[] p, bool raw = false)

Description: Sets the points of the given hole of the polygon

n: The index of the hole to which the points should be assigned

p: An array of points to assign to the polygon's hole

raw: If true, the points won't be compressed (see assign_hull)

If the hole index is not valid, this method does nothing.

This method was introduced in version 0.18. The 'raw' argument was added in version 0.24.

(2) Signature: void **assign_hole** (unsigned int n, const Box b)

Description: Sets the box as the given hole of the polygon

n: The index of the hole to which the points should be assigned

b: The box to assign to the polygon's hole

If the hole index is not valid, this method does nothing. This method was introduced in version 0.23.

assign_hull

Signature: void **assign_hull** (Point[] p, bool raw = false)

Description: Sets the points of the hull of polygon

p: An array of points to assign to the polygon's hull

raw: If true, the points won't be compressed

If the 'raw' argument is set to true, the points are taken as they are. Specifically no removal of redundant points or joining of coincident edges will take place. In effect, polygons consisting of a single point or two points can be constructed as well as polygons with duplicate points. Note that such polygons may cause problems in some applications.

Regardless of raw mode, the point list will be adjusted such that the first point is the lowest-leftmost one and the orientation is clockwise always.

The 'assign_hull' variant is provided in analogy to 'assign_hole'.

The 'raw' argument was added in version 0.24.

bbox

Signature: *[const]* Box **bbox**

Description: Returns the bounding box of the polygon

The bounding box is the box enclosing all points of the polygon.

break

Signature: *[const]* Polygon[] **break** (unsigned long max_vertex_count, double max_area_ratio)

Description: Splits the polygon into parts with a maximum vertex count and area ratio

The area ratio is the ratio between the bounding box area and the polygon area. Higher values mean more 'skinny' polygons.

This method will split the input polygon into pieces having a maximum of 'max_vertex_count' vertices and an area ratio less than 'max_area_ratio'. 'max_vertex_count' can be zero. In this case the limit is ignored. Also 'max_area_ratio' can be zero, in which case it is ignored as well.

The method of splitting is unspecified. The algorithm will apply 'split' recursively until the parts satisfy the limits.

This method has been introduced in version 0.29.

Python specific notes:
This attribute is available as 'break_' in Python.

compress

Signature: void **compress** (bool remove_reflected)

Description: Compresses the polygon.

remove_reflected: See description of the functionality.

This method removes redundant points from the polygon, such as points being on a line formed by two other points. If remove_reflected is true, points are also removed if the two adjacent edges form a spike.

This method was introduced in version 0.18.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

decompose_convex

Signature: *[const]* SimplePolygon[] **decompose_convex** (int preferred_orientation = PO_any)

Description: Decomposes the polygon into convex pieces

preferred_orientation: One of the PO_... constants

This method returns a decomposition of the polygon that contains convex pieces only. If the polygon was convex already, the list returned has a single element which is the original polygon.

This method was introduced in version 0.25.

decompose_trapezoids

Signature: *[const]* SimplePolygon[] **decompose_trapezoids** (int mode = TD_simple)

Description: Decomposes the polygon into trapezoids

mode: One of the TD_... constants

This method returns a decomposition of the polygon into trapezoid pieces. It supports different modes for various applications. See the TD_... constants for details.

This method was introduced in version 0.25.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Polygon ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_edge

(1) Signature: *[const,iter]* Edge **each_edge**

Description: Iterates over the edges that make up the polygon

This iterator will deliver all edges, including those of the holes. Hole edges are oriented counterclockwise while hull edges are oriented clockwise.

(2) Signature: *[const,iter]* Edge **each_edge** (unsigned int contour)

Description: Iterates over the edges of one contour of the polygon

contour: The contour number (0 for hull, 1 for first hole ...)

This iterator will deliver all edges of the contour specified by the contour parameter. The hull has contour number 0, the first hole has contour 1 etc. Hole edges are oriented counterclockwise while hull edges are oriented clockwise.

This method was introduced in version 0.24.

each_point_hole

Signature: *[const,iter]* Point **each_point_hole** (unsigned int n)

Description: Iterates over the points that make up the nth hole

The hole number must be less than the number of holes (see holes)

each_point_hull

Signature: *[const,iter]* Point **each_point_hull**

Description: Iterates over the points that make up the hull

ellipse

Signature: *[static]* new Polygon ptr **ellipse** (const Box box, int n)

Description: Creates a simple polygon approximating an ellipse

box: The bounding box of the ellipse

n: The number of points that will be used to approximate the ellipse

This method has been introduced in version 0.23.

extract_rad

Signature: *[const]* variant[] **extract_rad**

Description: Extracts the corner radii from a rounded polygon

Attempts to extract the radii of rounded corner polygon. This is essentially the inverse of the round_corners method. If this method succeeds, if will return an array of four elements:

- The polygon with the rounded corners replaced by edgy ones
- The radius of the inner corners
- The radius of the outer corners
- The number of points per full circle

This method is based on some assumptions and may fail. In this case, an empty array is returned.

If successful, the following code will more or less render the original polygon and parameters

```
p = ...   # some polygon
p.round_corners(ri, ro, n)
(p2, ri2, ro2, n2) = p.extract_rad
# -> p2 == p, ro2 == ro, ri2 == ri, n2 == n (within some limits)
```

This method was introduced in version 0.25.

from_dpoly

Signature: *[static]* new Polygon ptr **from_dpoly** (const DPolygon dpolygon)

Description: Creates an integer coordinate polygon from a floating-point coordinate polygon

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dpolygon'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new Polygon ptr **from_s** (string s)

Description: Creates a polygon from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given polygon. This method enables polygons as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

holes

Signature: *[const]* unsigned int **holes**

Description: Returns the number of holes

hull=

Signature: void **hull=** (Point[] p)

Description: Sets the points of the hull of polygon

p: An array of points to assign to the polygon's hull

The 'assign_hull' variant is provided in analogy to 'assign_hole'.

Python specific notes:
The object exposes a writable attribute 'hull'. This is the setter.

insert_hole

(1) Signature: void **insert_hole** (Point[] p, bool raw = false)

Description: Inserts a hole with the given points

p: An array of points to insert as a new hole

raw: If true, the points won't be compressed (see assign_hull)

The 'raw' argument was added in version 0.24.

(2) Signature: void **insert_hole** (const Box b)

Description: Inserts a hole from the given box

b: The box to insert as a new hole

This method was introduced in version 0.23.

inside?

Signature: *[const]* bool **inside?** (Point p)

Description: Tests, if the given point is inside the polygon

If the given point is inside or on the edge of the polygon, true is returned. This tests works well only if the polygon is not self-overlapping and oriented clockwise.

is_box?

Signature: *[const]* bool **is_box?**

Description: Returns true, if the polygon is a simple box.

Returns: True if the polygon is a box.

A polygon is a box if it is identical to its bounding box.

This method was introduced in version 0.23.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_convex?

Signature: *[const]* bool **is_convex?**

Description: Returns a value indicating whether the polygon is convex

This method will return true, if the polygon is convex.

This method was introduced in version 0.25.

is_empty?

Signature: *[const]* bool **is_empty?**

Description: Returns a value indicating whether the polygon is empty

is_halfmanhattan?

Signature: *[const]* bool **is_halfmanhattan?**

Description: Returns a value indicating whether the polygon is half-manhattan

Half-manhattan polygons have edges which are multiples of 45 degree. These polygons can be clipped at a rectangle without potential grid snapping.

This predicate was introduced in version 0.27.

is_rectilinear?

Signature: *[const]* bool **is_rectilinear?**

Description: Returns a value indicating whether the polygon is rectilinear

minkowski_sum

(1) Signature: *[const]* Polygon **minkowski_sum** (const Edge e, bool resolve_holes)

Description: Computes the Minkowski sum of the polygon and an edge

e:	The edge.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum with the edge e.

The Minkowski sum of a polygon and an edge basically results in the area covered when "dragging" the polygon along the line given by the edge. The effect is similar to drawing the line with a pencil that has the shape of the given polygon.

This method was introduced in version 0.22.

(2) Signature: *[const]* Polygon **minkowski_sum** (const Polygon b, bool resolve_holes)

Description: Computes the Minkowski sum of the polygon and a polygon

p:	The first argument.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and p.

This method was introduced in version 0.22.

(3) Signature: *[const]* Polygon **minkowski_sum** (const Box b, bool resolve_holes)

Description: Computes the Minkowski sum of the polygon and a box

b:	The box.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and the box.

This method was introduced in version 0.22.

(4) Signature: *[const]* Polygon **minkowski_sum** (Point[] b, bool resolve_holes)

Description: Computes the Minkowski sum of the polygon and a contour of points (a trace)

b:	The contour (a series of points forming the trace).
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and the contour.

This method was introduced in version 0.22.

minkowsky_sum

(1) Signature: *[const]* Polygon **minkowsky_sum** (const Edge e, bool resolve_holes)

Description: Computes the Minkowski sum of the polygon and an edge

e:	The edge.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum with the edge e.

Use of this method is deprecated. Use minkowski_sum instead

The Minkowski sum of a polygon and an edge basically results in the area covered when "dragging" the polygon along the line given by the edge. The effect is similar to drawing the line with a pencil that has the shape of the given polygon.

This method was introduced in version 0.22.

(2) Signature: *[const]* Polygon **minkowsky_sum** (const Polygon b, bool resolve_holes)

Description: Computes the Minkowski sum of the polygon and a polygon

p:	The first argument.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and p.

Use of this method is deprecated. Use minkowski_sum instead

This method was introduced in version 0.22.

(3) Signature: *[const]* Polygon **minkowsky_sum** (const Box b, bool resolve_holes)

Description: Computes the Minkowski sum of the polygon and a box

b:	The box.
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and the box.

Use of this method is deprecated. Use minkowski_sum instead

This method was introduced in version 0.22.

(4) Signature: *[const]* Polygon **minkowsky_sum** (Point[] b, bool resolve_holes)

Description: Computes the Minkowski sum of the polygon and a contour of points (a trace)

b:	The contour (a series of points forming the trace).
resolve_holes:	If true, the output polygon will not contain holes, but holes are resolved by joining the holes with the hull.
Returns:	The new polygon representing the Minkowski sum of self and the contour.

Use of this method is deprecated. Use minkowski_sum instead

This method was introduced in version 0.22.

move

(1) Signature: Polygon **move** (const Vector p)

Description: Moves the polygon.

p:	The distance to move the polygon.
Returns:	The moved polygon (self).

Moves the polygon by the given offset and returns the moved polygon. The polygon is overwritten.

This method has been introduced in version 0.23.

(2) Signature: Polygon **move** (int x, int y)

Description: Moves the polygon.

x:	The x distance to move the polygon.
y:	The y distance to move the polygon.
Returns:	The moved polygon (self).

Moves the polygon by the given offset and returns the moved polygon. The polygon is overwritten.

moved

(1) Signature: *[const]* Polygon **moved** (const Vector p)

Description: Returns the moved polygon (does not modify self)

p:	The distance to move the polygon.
Returns:	The moved polygon.

Moves the polygon by the given offset and returns the moved polygon. The polygon is not modified.

This method has been introduced in version 0.23.

(2) Signature: *[const]* Polygon **moved** (int x, int y)

Description: Returns the moved polygon (does not modify self)

x:	The x distance to move the polygon.
y:	The y distance to move the polygon.
Returns:	The moved polygon.

Moves the polygon by the given offset and returns the moved polygon. The polygon is not modified.

This method has been introduced in version 0.23.

new

(1) Signature: *[static]* new Polygon ptr **new** (const DPolygon dpolygon)

Description: Creates an integer coordinate polygon from a floating-point coordinate polygon

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dpolygon'.

Python specific notes:

This method is the default initializer of the object.

(2) Signature: *[static]* new Polygon ptr **new**

Description: Creates an empty (invalid) polygon

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Polygon ptr **new** (const SimplePolygon sp)

Description: Creates a polygon from a simple polygon

sp: The simple polygon that is converted into the polygon

This method was introduced in version 0.22.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Polygon ptr **new** (Point[] pts, bool raw = false)

Description: Creates a polygon from a point array for the hull

pts: The points forming the polygon hull

raw: If true, the point list won't be modified (see assign_hull)

The 'raw' argument was added in version 0.24.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new Polygon ptr **new** (const Box box)

Description: Creates a polygon from a box

box: The box to convert to a polygon

Python specific notes:
This method is the default initializer of the object.

num_points

Signature: *[const]* unsigned long **num_points**

Description: Gets the total number of points (hull plus holes)

This method was introduced in version 0.18.

num_points_hole

Signature: *[const]* unsigned long **num_points_hole** (unsigned int n)

Description: Gets the number of points of the given hole

The argument gives the index of the hole of which the number of points are requested. The index must be less than the number of holes (see holes).

num_points_hull

Signature: *[const]* unsigned long **num_points_hull**

Description: Gets the number of points of the hull

perimeter

Signature: *[const]* unsigned long **perimeter**

Description: Gets the perimeter of the polygon

The perimeter is sum of the lengths of all edges making up the polygon.

This method has been introduce in version 0.23.

point_hole

Signature: *[const]* Point **point_hole** (unsigned int n, unsigned long p)

Description: Gets a specific point of a hole

n: The index of the hole to which the points should be assigned

p: The index of the point to get

If the index of the point or of the hole is not valid, a default value is returned. This method was introduced in version 0.18.

point_hull

Signature: *[const]* Point **point_hull** (unsigned long p)

Description: Gets a specific point of the hull

p: The index of the point to get

If the index of the point is not a valid index, a default value is returned. This method was introduced in version 0.18.

resolve_holes

Signature: void **resolve_holes**

Description: Resolve holes by inserting cut lines and joining the holes with the hull

This method modifies the polygon. The out-of-place version is resolved_holes. This method was introduced in version 0.22.

resolved_holes

Signature: *[const]* Polygon **resolved_holes**

Description: Returns a polygon without holes

Returns: The new polygon without holes.

This method does not modify the polygon but return a new polygon. This method was introduced in version 0.22.

round_corners

Signature: *[const]* Polygon **round_corners** (double rinner, double router, unsigned int n)

Description: Rounds the corners of the polygon

rinner: The circle radius of inner corners (in database units).

router: The circle radius of outer corners (in database units).

n: The number of points per full circle.

Returns: The new polygon.

Replaces the corners of the polygon with circle segments.

This method was introduced in version 0.20 for integer coordinates and in 0.25 for all coordinate types.

size

(1) Signature: void **size** (int dx, int dy, unsigned int mode)

Description: Sizes the polygon (biasing)

Shifts the contour outwards (dx,dy>0) or inwards (dx,dy<0). dx is the sizing in x-direction and dy is the sizing in y-direction. The sign of dx and dy should be identical. The sizing operation create invalid (self-overlapping, reverse oriented) contours.

The mode defines at which bending angle cutoff occurs (0:>0, 1:>45, 2:>90, 3:>135, 4:>approx. 168, other:>approx. 179)

In order to obtain a proper polygon in the general case, the sized polygon must be merged in 'greater than zero' wrap count mode. This is necessary since in the general case, sizing can be complicated

operation which lets a single polygon fall apart into disjoint pieces for example. This can be achieved using the EdgeProcessor class for example:

```
poly = ... # a RBA::Polygon
poly.size(-50, 2)
ep = RBA::EdgeProcessor::new
# result is an array of RBA::Polygon objects
result = ep.simple_merge_p2p([ poly ], false, false, 1)
```

(2) Signature: void **size** (const Vector dv, unsigned int mode = 2)

Description: Sizes the polygon (biasing)

This method is equivalent to

```
size(dv.x, dv.y, mode)
```

See size for a detailed description.

This version has been introduced in version 0.28.

(3) Signature: void **size** (int d, unsigned int mode = 2)

Description: Sizes the polygon (biasing)

Shifts the contour outwards (d>0) or inwards (d<0). This method is equivalent to

```
size(d, d, mode)
```

See size for a detailed description.

This method has been introduced in version 0.23.

sized

(1) Signature: *[const]* Polygon **sized** (int dx, int dy, unsigned int mode)

Description: Sizes the polygon (biasing) without modifying self

This method applies sizing to the polygon but does not modify self. Instead a sized copy is returned. See size for a description of the operation.

This method has been introduced in version 0.23.

(2) Signature: *[const]* Polygon **sized** (const Vector dv, unsigned int mode = 2)

Description: Sizes the polygon (biasing) without modifying self

This method is equivalent to

```
sized(dv.x, dv.y, mode)
```

See size and sized for a detailed description.

This version has been introduced in version 0.28.

(3) Signature: *[const]* Polygon **sized** (int d, unsigned int mode = 2)

Description: Sizes the polygon (biasing) without modifying self

Shifts the contour outwards (d>0) or inwards (d<0). This method is equivalent to

```
sized(d, d, mode)
```

See size and sized for a detailed description.

smooth

Signature: *[const]* Polygon **smooth** (int d, bool keep_hv = false)

Description: Smooths a polygon

d:	The smoothing "roughness".
keep_hv:	If true, horizontal and vertical edges will be preserved always.
Returns:	The smoothed polygon.

Remove vertices that deviate by more than the distance d from the average contour. The value d is basically the roughness which is removed.

This method was introduced in version 0.23. The 'keep_hv' optional parameter was added in version 0.27.

sort_holes

Signature: void **sort_holes**

Description: Brings the holes in a specific order

This function is normalize the hole order so the comparison of two polygons does not depend on the order the holes were inserted. Polygons generated by KLayout's alorithms have their holes sorted.

This method has been introduced in version 0.28.8.

split

Signature: *[const]* Polygon[] **split**

Description: Splits the polygon into two or more parts

This method will break the polygon into parts. The exact breaking algorithm is unspecified, the result are smaller polygons of roughly equal number of points and 'less concave' nature. Usually the returned polygon set consists of two polygons, but there can be more. The merged region of the resulting polygons equals the original polygon with the exception of small snapping effects at new vertexes.

The intended use for this method is a iteratively split polygons until the satisfy some maximum number of points limit.

This method has been introduced in version 0.25.3.

to_dtype

Signature: *[const]* DPolygon **to_dtype** (double dbu = 1)

Description: Converts the polygon to a floating-point coordinate polygon

The database unit can be specified to translate the integer-coordinate polygon into a floating-point coordinate polygon in micron units. The database unit is basically a scaling factor.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s**

Description: Returns a string representing the polygon

Python specific notes:
This method is also available as 'str(object)'.

to_simple_polygon

Signature: *[const]* SimplePolygon **to_simple_polygon**

Description: Converts a polygon to a simple polygon

Returns: The simple polygon.

If the polygon contains holes, these will be resolved. This operation requires a well-formed polygon. Reflecting edges, self-intersections and coincident points will be removed.

This method was introduced in version 0.22.

touches?

(1) Signature: *[const]* bool **touches?** (const Box box)

Description: Returns true, if the polygon touches the given box.

The box and the polygon touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

(2) Signature: *[const]* bool **touches?** (const Edge edge)

Description: Returns true, if the polygon touches the given edge.

The edge and the polygon touch if they overlap or the edge shares at least one point with the polygon's contour.

This method was introduced in version 0.25.1.

(3) Signature: *[const]* bool **touches?** (const Polygon polygon)

Description: Returns true, if the polygon touches the other polygon.

The polygons touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

(4) Signature: *[const]* bool **touches?** (const SimplePolygon simple_polygon)

Description: Returns true, if the polygon touches the other polygon.

The polygons touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

transform

(1) Signature: Polygon ptr **transform** (const ICplxTrans t)

Description: Transforms the polygon with a complex transformation (in-place)

t: The transformation to apply.

Transforms the polygon with the given complex transformation. This version modifies self and will return self as the modified polygon. An out-of-place version which does not modify self is transformed.

This method was introduced in version 0.24.

(2) Signature: Polygon ptr **transform** (const Trans t)

Description: Transforms the polygon (in-place)

t: The transformation to apply.

Transforms the polygon with the given transformation. Modifies self and returns self. An out-of-place version which does not modify self is transformed.

This method has been introduced in version 0.24.

transformed

(1) Signature: *[const]* Polygon **transformed** (const ICplxTrans t)

Description: Transforms the polygon with a complex transformation

t:	The transformation to apply.
Returns:	The transformed polygon (in this case an integer coordinate polygon).

Use of this method is deprecated

Transforms the polygon with the given complex transformation. Does not modify the polygon but returns the transformed polygon.

This method was introduced in version 0.18.

(2) Signature: *[const]* Polygon **transformed** (const Trans t)

Description: Transforms the polygon

t:	The transformation to apply.
Returns:	The transformed polygon.

Transforms the polygon with the given transformation. Does not modify the polygon but returns the transformed polygon.

(3) Signature: *[const]* DPolygon **transformed** (const CplxTrans t)

Description: Transforms the polygon with a complex transformation

t:	The transformation to apply.
Returns:	The transformed polygon.

Transforms the polygon with the given complex transformation. Does not modify the polygon but returns the transformed polygon.

With version 0.25, the original 'transformed_cplx' method is deprecated and 'transformed' takes both simple and complex transformations.

transformed_cplx

Signature: *[const]* DPolygon **transformed_cplx** (const CplxTrans t)

Description: Transforms the polygon with a complex transformation

t:	The transformation to apply.
Returns:	The transformed polygon.

Use of this method is deprecated. Use transformed instead

Transforms the polygon with the given complex transformation. Does not modify the polygon but returns the transformed polygon.

With version 0.25, the original 'transformed_cplx' method is deprecated and 'transformed' takes both simple and complex transformations.

4.77. API reference - Class DPolygon

Notation used in Ruby API documentation

Module: db

Description: A polygon class

A polygon consists of an outer hull and zero to many holes. Each contour consists of several points. The point list is normalized such that the leftmost, lowest point is the first one. The orientation is normalized such that the orientation of the hull contour is clockwise, while the orientation of the holes is counterclockwise.

It is in no way checked that the contours are not overlapping. This must be ensured by the user of the object when filling the contours.

A polygon can be asked for the number of holes using the holes method. each_point_hull delivers the points of the hull contour. each_point_hole delivers the points of a specific hole. each_edge delivers the edges (point-to-point connections) of both hull and holes. bbox delivers the bounding box, area the area and perimeter the perimeter of the polygon.

Here's an example of how to create a polygon:

```
hull =  [ RBA::DPoint::new(0, 0),       RBA::DPoint::new(6000, 0),
          RBA::DPoint::new(6000, 3000), RBA::DPoint::new(0, 3000) ]
hole1 = [ RBA::DPoint::new(1000, 1000), RBA::DPoint::new(2000, 1000),
          RBA::DPoint::new(2000, 2000), RBA::DPoint::new(1000, 2000) ]
hole2 = [ RBA::DPoint::new(3000, 1000), RBA::DPoint::new(4000, 1000),
          RBA::DPoint::new(4000, 2000), RBA::DPoint::new(3000, 2000) ]
poly = RBA::DPolygon::new(hull)
poly.insert_hole(hole1)
poly.insert_hole(hole2)

# ask the polygon for some properties
poly.holes      # -> 2
poly.area       # -> 16000000.0
poly.perimeter  # -> 26000.0
poly.bbox       # -> (0,0;6000,3000)
```

The DPolygon class stores coordinates in floating-point format which gives a higher precision for some operations. A class that stores integer coordinates is Polygon.

See The Database API for more details about the database objects.

Public constructors

new DPolygon ptr	**new**	(const Polygon polygon)	Creates a floating-point coordinate polygon from an integer coordinate polygon
new DPolygon ptr	**new**		Creates an empty (invalid) polygon
new DPolygon ptr	**new**	(const DSimplePolygon sp)	Creates a polygon from a simple polygon
new DPolygon ptr	**new**	(DPoint[] pts, bool raw = false)	Creates a polygon from a point array for the hull
new DPolygon ptr	**new**	(const DBox box)	Creates a polygon from a box

Public methods

[const]	bool	**!=**	(const DPolygon p)	Returns a value indicating whether the polygons are not equal

[const]	DPolygon	*****	(double f)	Scales the polygon by some factor
[const]	bool	**<**	(const DPolygon p)	Returns a value indicating whether self is less than p
[const]	bool	**==**	(const DPolygon p)	Returns a value indicating whether the polygons are equal
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	double	**area**		Gets the area of the polygon
[const]	double	**area2**		Gets the double area of the polygon
	void	**assign**	(const DPolygon other)	Assigns another object to self
	void	**assign_hole**	(unsigned int n, DPoint[] p, bool raw = false)	Sets the points of the given hole of the polygon
	void	**assign_hole**	(unsigned int n, const DBox b)	Sets the box as the given hole of the polygon
	void	**assign_hull**	(DPoint[] p, bool raw = false)	Sets the points of the hull of polygon
[const]	DBox	**bbox**		Returns the bounding box of the polygon
[const]	DPolygon[]	**break**	(unsigned long max_vertex_count, double max_area_ratio)	Splits the polygon into parts with a maximum vertex count and area ratio
	void	**compress**	(bool remove_reflected)	Compresses the polygon.
[const]	new DPolygon ptr	**dup**		Creates a copy of self
[const,iter]	DEdge	**each_edge**		Iterates over the edges that make up the polygon
[const,iter]	DEdge	**each_edge**	(unsigned int contour)	Iterates over the edges of one contour of the polygon

[const,iter]	DPoint	**each_point_hole**	(unsigned int n)	Iterates over the points that make up the nth hole
[const,iter]	DPoint	**each_point_hull**		Iterates over the points that make up the hull
[const]	variant[]	**extract_rad**		Extracts the corner radii from a rounded polygon
[const]	unsigned long	**hash**		Computes a hash value
[const]	unsigned int	**holes**		Returns the number of holes
	void	**hull=**	(DPoint[] p)	Sets the points of the hull of polygon
	void	**insert_hole**	(DPoint[] p, bool raw = false)	Inserts a hole with the given points
	void	**insert_hole**	(const DBox b)	Inserts a hole from the given box
[const]	bool	**inside?**	(DPoint p)	Tests, if the given point is inside the polygon
[const]	bool	**is_box?**		Returns true, if the polygon is a simple box.
[const]	bool	**is_empty?**		Returns a value indicating whether the polygon is empty
[const]	bool	**is_halfmanhattan?**		Returns a value indicating whether the polygon is half-manhattan
[const]	bool	**is_rectilinear?**		Returns a value indicating whether the polygon is rectilinear
	DPolygon	**move**	(const DVector p)	Moves the polygon.
	DPolygon	**move**	(double x, double y)	Moves the polygon.
[const]	DPolygon	**moved**	(const DVector p)	Returns the moved polygon (does not modify self)
[const]	DPolygon	**moved**	(double x, double y)	Returns the moved polygon (does not modify self)
[const]	unsigned long	**num_points**		Gets the total number of points (hull plus holes)
[const]	unsigned long	**num_points_hole**	(unsigned int n)	Gets the number of points of the given hole
[const]	unsigned long	**num_points_hull**		Gets the number of points of the hull
[const]	double	**perimeter**		Gets the perimeter of the polygon
[const]	DPoint	**point_hole**	(unsigned int n, unsigned long p)	Gets a specific point of a hole
[const]	DPoint	**point_hull**	(unsigned long p)	Gets a specific point of the hull

[const]	DPolygon	**round_corners**	(double rinner, double router, unsigned int n)	Rounds the corners of the polygon
	void	**size**	(double dx, double dy, unsigned int mode)	Sizes the polygon (biasing)
	void	**size**	(const Vector dv, unsigned int mode = 2)	Sizes the polygon (biasing)
	void	**size**	(double d, unsigned int mode = 2)	Sizes the polygon (biasing)
[const]	DPolygon	**sized**	(double dx, double dy, unsigned int mode)	Sizes the polygon (biasing) without modifying self
[const]	DPolygon	**sized**	(const Vector dv, unsigned int mode = 2)	Sizes the polygon (biasing) without modifying self
[const]	DPolygon	**sized**	(double d, unsigned int mode = 2)	Sizes the polygon (biasing) without modifying self
	void	**sort_holes**		Brings the holes in a specific order
[const]	DPolygon[]	**split**		Splits the polygon into two or more parts
[const]	Polygon	**to_itype**	(double dbu = 1)	Converts the polygon to an integer coordinate polygon
[const]	string	**to_s**		Returns a string representing the polygon
[const]	bool	**touches?**	(const DBox box)	Returns true, if the polygon touches the given box.
[const]	bool	**touches?**	(const DEdge edge)	Returns true, if the polygon touches the given edge.
[const]	bool	**touches?**	(const DPolygon polygon)	Returns true, if the polygon touches the other polygon.
[const]	bool	**touches?**	(const DSimplePolygon simple_polygon)	Returns true, if the polygon touches the other polygon.
	DPolygon ptr	**transform**	(const DCplxTrans t)	Transforms the polygon with a complex transformation (in-place)
	DPolygon ptr	**transform**	(const DTrans t)	Transforms the polygon (in-place)
[const]	Polygon	**transformed**	(const VCplxTrans t)	Transforms the polygon with the given complex transformation
[const]	DPolygon	**transformed**	(const DTrans t)	Transforms the polygon

[const]	DPolygon	**transformed**	(const DCplxTrans t)	Transforms the polygon with a complex transformation

Public static methods and constants

	new DPolygon ptr	**ellipse**	(const DBox box, int n)	Creates a simple polygon approximating an ellipse
	new DPolygon ptr	**from_s**	(string s)	Creates a polygon from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new DPolygon ptr	**from_ipoly**	(const Polygon polygon)	Use of this method is deprecated. Use new instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	DPolygon	**transformed_cplx**	(const DCplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

!=

Signature: *[const]* bool **!=** (const DPolygon p)

Description: Returns a value indicating whether the polygons are not equal

p: The object to compare against

*****

Signature: *[const]* DPolygon ***** (double f)

Description: Scales the polygon by some factor

Returns the scaled object. All coordinates are multiplied with the given factor and if necessary rounded.

Python specific notes:
This method also implements '__rmul__'.

<

Signature: *[const]* bool **<** (const DPolygon p)

Description: Returns a value indicating whether self is less than p

p: The object to compare against

This operator is provided to establish some, not necessarily a certain sorting order

==

Signature: *[const]* bool **==** (const DPolygon p)

Description: Returns a value indicating whether the polygons are equal

p: The object to compare against

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

area

Signature: *[const]* double **area**

Description: Gets the area of the polygon

The area is correct only if the polygon is not self-overlapping and the polygon is oriented clockwise.Orientation is ensured automatically in most cases.

area2

Signature: *[const]* double **area2**

Description: Gets the double area of the polygon

This method is provided because the area for an integer-type polygon is a multiple of 1/2. Hence the double area can be expresses precisely as an integer for these types.

This method has been introduced in version 0.26.1

assign

Signature: void **assign** (const DPolygon other)

Description: Assigns another object to self

assign_hole

(1) Signature: void **assign_hole** (unsigned int n, DPoint[] p, bool raw = false)

Description: Sets the points of the given hole of the polygon

n:	The index of the hole to which the points should be assigned
p:	An array of points to assign to the polygon's hole
raw:	If true, the points won't be compressed (see assign_hull)

If the hole index is not valid, this method does nothing.

This method was introduced in version 0.18. The 'raw' argument was added in version 0.24.

(2) Signature: void **assign_hole** (unsigned int n, const DBox b)

Description: Sets the box as the given hole of the polygon

n:	The index of the hole to which the points should be assigned
b:	The box to assign to the polygon's hole

If the hole index is not valid, this method does nothing. This method was introduced in version 0.23.

assign_hull

Signature: void **assign_hull** (DPoint[] p, bool raw = false)

Description: Sets the points of the hull of polygon

p:	An array of points to assign to the polygon's hull
raw:	If true, the points won't be compressed

If the 'raw' argument is set to true, the points are taken as they are. Specifically no removal of redundant points or joining of coincident edges will take place. In effect, polygons consisting of a single point or two points can be constructed as well as polygons with duplicate points. Note that such polygons may cause problems in some applications.

Regardless of raw mode, the point list will be adjusted such that the first point is the lowest-leftmost one and the orientation is clockwise always.

The 'assign_hull' variant is provided in analogy to 'assign_hole'.

The 'raw' argument was added in version 0.24.

bbox

Signature: *[const]* DBox **bbox**

Description: Returns the bounding box of the polygon

The bounding box is the box enclosing all points of the polygon.

break

Signature: *[const]* DPolygon[] **break** (unsigned long max_vertex_count, double max_area_ratio)

Description: Splits the polygon into parts with a maximum vertex count and area ratio

The area ratio is the ratio between the bounding box area and the polygon area. Higher values mean more 'skinny' polygons.

This method will split the input polygon into pieces having a maximum of 'max_vertex_count' vertices and an area ratio less than 'max_area_ratio'. 'max_vertex_count' can be zero. In this case the limit is ignored. Also 'max_area_ratio' can be zero, in which case it is ignored as well.

The method of splitting is unspecified. The algorithm will apply 'split' recursively until the parts satisfy the limits.

This method has been introduced in version 0.29.

Python specific notes:
This attribute is available as 'break_' in Python.

compress

Signature: void **compress** (bool remove_reflected)

Description: Compresses the polygon.

remove_reflected: See description of the functionality.

This method removes redundant points from the polygon, such as points being on a line formed by two other points. If remove_reflected is true, points are also removed if the two adjacent edges form a spike.

This method was introduced in version 0.18.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new DPolygon ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_edge

(1) Signature: *[const,iter]* DEdge **each_edge**

Description: Iterates over the edges that make up the polygon

This iterator will deliver all edges, including those of the holes. Hole edges are oriented counterclockwise while hull edges are oriented clockwise.

(2) Signature: *[const,iter]* DEdge **each_edge** (unsigned int contour)

Description: Iterates over the edges of one contour of the polygon

contour: The contour number (0 for hull, 1 for first hole ...)

This iterator will deliver all edges of the contour specified by the contour parameter. The hull has contour number 0, the first hole has contour 1 etc. Hole edges are oriented counterclockwise while hull edges are oriented clockwise.

This method was introduced in version 0.24.

each_point_hole

Signature: *[const,iter]* DPoint **each_point_hole** (unsigned int n)

Description: Iterates over the points that make up the nth hole

The hole number must be less than the number of holes (see holes)

each_point_hull

Signature: *[const,iter]* DPoint **each_point_hull**

Description: Iterates over the points that make up the hull

ellipse

Signature: *[static]* new DPolygon ptr **ellipse** (const DBox box, int n)

Description: Creates a simple polygon approximating an ellipse

box: The bounding box of the ellipse

n: The number of points that will be used to approximate the ellipse

This method has been introduced in version 0.23.

extract_rad

Signature: *[const]* variant[] **extract_rad**

Description: Extracts the corner radii from a rounded polygon

Attempts to extract the radii of rounded corner polygon. This is essentially the inverse of the round_corners method. If this method succeeds, if will return an array of four elements:

- The polygon with the rounded corners replaced by edgy ones
- The radius of the inner corners
- The radius of the outer corners
- The number of points per full circle

This method is based on some assumptions and may fail. In this case, an empty array is returned.

If successful, the following code will more or less render the original polygon and parameters

```
p = ...   # some polygon
p.round_corners(ri, ro, n)
(p2, ri2, ro2, n2) = p.extract_rad
# -> p2 == p, ro2 == ro, ri2 == ri, n2 == n (within some limits)
```

This method was introduced in version 0.25.

from_ipoly

Signature: *[static]* new DPolygon ptr **from_ipoly** (const Polygon polygon)

Description: Creates a floating-point coordinate polygon from an integer coordinate polygon

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_ipolygon'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new DPolygon ptr **from_s** (string s)

Description: Creates a polygon from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given polygon. This method enables polygons as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

holes

Signature: *[const]* unsigned int **holes**

Description: Returns the number of holes

hull=

Signature: void **hull=** (DPoint[] p)

Description: Sets the points of the hull of polygon

p: An array of points to assign to the polygon's hull

The 'assign_hull' variant is provided in analogy to 'assign_hole'.

Python specific notes:
The object exposes a writable attribute 'hull'. This is the setter.

insert_hole

(1) Signature: void **insert_hole** (DPoint[] p, bool raw = false)

Description: Inserts a hole with the given points

p: An array of points to insert as a new hole

raw: If true, the points won't be compressed (see assign_hull)

The 'raw' argument was added in version 0.24.

(2) Signature: void **insert_hole** (const DBox b)

Description: Inserts a hole from the given box

b: The box to insert as a new hole

This method was introduced in version 0.23.

inside?

Signature: *[const]* bool **inside?** (DPoint p)

Description: Tests, if the given point is inside the polygon

If the given point is inside or on the edge of the polygon, true is returned. This tests works well only if the polygon is not self-overlapping and oriented clockwise.

is_box?

Signature: *[const]* bool **is_box?**

Description: Returns true, if the polygon is a simple box.

Returns: True if the polygon is a box.

A polygon is a box if it is identical to its bounding box.

This method was introduced in version 0.23.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_empty?

Signature: *[const]* bool **is_empty?**

Description: Returns a value indicating whether the polygon is empty

is_halfmanhattan?

Signature: *[const]* bool **is_halfmanhattan?**

Description: Returns a value indicating whether the polygon is half-manhattan

Half-manhattan polygons have edges which are multiples of 45 degree. These polygons can be clipped at a rectangle without potential grid snapping.

This predicate was introduced in version 0.27.

is_rectilinear?

Signature: *[const]* bool **is_rectilinear?**

Description: Returns a value indicating whether the polygon is rectilinear

move

(1) Signature: DPolygon **move** (const DVector p)

Description: Moves the polygon.

p: The distance to move the polygon.

Returns: The moved polygon (self).

Moves the polygon by the given offset and returns the moved polygon. The polygon is overwritten.

This method has been introduced in version 0.23.

(2) Signature: DPolygon **move** (double x, double y)

Description: Moves the polygon.

x: The x distance to move the polygon.

y: The y distance to move the polygon.

Returns: The moved polygon (self).

Moves the polygon by the given offset and returns the moved polygon. The polygon is overwritten.

moved

(1) Signature: *[const]* DPolygon **moved** (const DVector p)

Description: Returns the moved polygon (does not modify self)

p: The distance to move the polygon.

Returns: The moved polygon.

Moves the polygon by the given offset and returns the moved polygon. The polygon is not modified.

This method has been introduced in version 0.23.

(2) Signature: *[const]* DPolygon **moved** (double x, double y)

Description: Returns the moved polygon (does not modify self)

x: The x distance to move the polygon.

y: The y distance to move the polygon.

Returns: The moved polygon.

Moves the polygon by the given offset and returns the moved polygon. The polygon is not modified.

This method has been introduced in version 0.23.

new

(1) Signature: *[static]* new DPolygon ptr **new** (const Polygon polygon)

Description: Creates a floating-point coordinate polygon from an integer coordinate polygon

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_ipolygon'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DPolygon ptr **new**

Description: Creates an empty (invalid) polygon

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new DPolygon ptr **new** (const DSimplePolygon sp)

Description: Creates a polygon from a simple polygon

sp: The simple polygon that is converted into the polygon

This method was introduced in version 0.22.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new DPolygon ptr **new** (DPoint[] pts, bool raw = false)

Description: Creates a polygon from a point array for the hull

pts: The points forming the polygon hull

raw: If true, the point list won't be modified (see assign_hull)

The 'raw' argument was added in version 0.24.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new DPolygon ptr **new** (const DBox box)

Description: Creates a polygon from a box

box: The box to convert to a polygon

Python specific notes:

This method is the default initializer of the object.

num_points

Signature: *[const]* unsigned long **num_points**

Description: Gets the total number of points (hull plus holes)

This method was introduced in version 0.18.

num_points_hole

Signature: *[const]* unsigned long **num_points_hole** (unsigned int n)

Description: Gets the number of points of the given hole

The argument gives the index of the hole of which the number of points are requested. The index must be less than the number of holes (see holes).

num_points_hull

Signature: *[const]* unsigned long **num_points_hull**

Description: Gets the number of points of the hull

perimeter

Signature: *[const]* double **perimeter**

Description: Gets the perimeter of the polygon

The perimeter is sum of the lengths of all edges making up the polygon.

This method has been introduce in version 0.23.

point_hole

Signature: *[const]* DPoint **point_hole** (unsigned int n, unsigned long p)

Description: Gets a specific point of a hole

n: The index of the hole to which the points should be assigned

p: The index of the point to get

If the index of the point or of the hole is not valid, a default value is returned. This method was introduced in version 0.18.

point_hull

Signature: *[const]* DPoint **point_hull** (unsigned long p)

Description: Gets a specific point of the hull

p: The index of the point to get

If the index of the point is not a valid index, a default value is returned. This method was introduced in version 0.18.

round_corners

Signature: *[const]* DPolygon **round_corners** (double rinner, double router, unsigned int n)

Description: Rounds the corners of the polygon

rinner: The circle radius of inner corners (in database units).

router: The circle radius of outer corners (in database units).

n: The number of points per full circle.

Returns: The new polygon.

Replaces the corners of the polygon with circle segments.

This method was introduced in version 0.20 for integer coordinates and in 0.25 for all coordinate types.

size

(1) Signature: void **size** (double dx, double dy, unsigned int mode)

Description: Sizes the polygon (biasing)

Shifts the contour outwards (dx,dy>0) or inwards (dx,dy<0). dx is the sizing in x-direction and dy is the sizing in y-direction. The sign of dx and dy should be identical. The sizing operation create invalid (self-overlapping, reverse oriented) contours.

The mode defines at which bending angle cutoff occurs (0:>0, 1:>45, 2:>90, 3:>135, 4:>approx. 168, other:>approx. 179)

In order to obtain a proper polygon in the general case, the sized polygon must be merged in 'greater than zero' wrap count mode. This is necessary since in the general case, sizing can be complicated operation which lets a single polygon fall apart into disjoint pieces for example. This can be achieved using the EdgeProcessor class for example:

```
poly = ... # a RBA::Polygon
poly.size(-50, 2)
ep = RBA::EdgeProcessor::new
# result is an array of RBA::Polygon objects
result = ep.simple_merge_p2p([ poly ], false, false, 1)
```

(2) Signature: void **size** (const Vector dv, unsigned int mode = 2)

Description: Sizes the polygon (biasing)

This method is equivalent to

```
size(dv.x, dv.y, mode)
```

See size for a detailed description.

This version has been introduced in version 0.28.

(3) Signature: void **size** (double d, unsigned int mode = 2)

Description: Sizes the polygon (biasing)

Shifts the contour outwards (d>0) or inwards (d<0). This method is equivalent to

```
size(d, d, mode)
```

See size for a detailed description.

This method has been introduced in version 0.23.

sized

(1) Signature: *[const]* DPolygon **sized** (double dx, double dy, unsigned int mode)

Description: Sizes the polygon (biasing) without modifying self

This method applies sizing to the polygon but does not modify self. Instead a sized copy is returned. See size for a description of the operation.

This method has been introduced in version 0.23.

(2) Signature: *[const]* DPolygon **sized** (const Vector dv, unsigned int mode = 2)

Description: Sizes the polygon (biasing) without modifying self

This method is equivalent to

```
sized(dv.x, dv.y, mode)
```

See size and sized for a detailed description.

This version has been introduced in version 0.28.

(3) Signature: *[const]* DPolygon **sized** (double d, unsigned int mode = 2)

Description: Sizes the polygon (biasing) without modifying self

Shifts the contour outwards (d>0) or inwards (d<0). This method is equivalent to

```
sized(d, d, mode)
```

See size and sized for a detailed description.

sort_holes

Signature: void **sort_holes**

Description: Brings the holes in a specific order

This function is normalize the hole order so the comparison of two polygons does not depend on the order the holes were inserted. Polygons generated by KLayout's alorithms have their holes sorted.

This method has been introduced in version 0.28.8.

split

Signature: *[const]* DPolygon[] **split**

Description: Splits the polygon into two or more parts

This method will break the polygon into parts. The exact breaking algorithm is unspecified, the result are smaller polygons of roughly equal number of points and 'less concave' nature. Usually the returned polygon set consists of two polygons, but there can be more. The merged region of the resulting polygons equals the original polygon with the exception of small snapping effects at new vertexes.

The intended use for this method is a iteratively split polygons until the satisfy some maximum number of points limit.

This method has been introduced in version 0.25.3.

to_itype

Signature: *[const]* Polygon **to_itype** (double dbu = 1)

Description: Converts the polygon to an integer coordinate polygon

The database unit can be specified to translate the floating-point coordinate polygon in micron units to an integer-coordinate polygon in database units. The polygons coordinates will be divided by the database unit.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s**

Description: Returns a string representing the polygon

Python specific notes:
This method is also available as 'str(object)'.

touches?

(1) Signature: *[const]* bool **touches?** (const DBox box)

Description: Returns true, if the polygon touches the given box.

The box and the polygon touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

(2) Signature: *[const]* bool **touches?** (const DEdge edge)

Description: Returns true, if the polygon touches the given edge.

The edge and the polygon touch if they overlap or the edge shares at least one point with the polygon's contour.

This method was introduced in version 0.25.1.

(3) Signature: *[const]* bool **touches?** (const DPolygon polygon)

Description: Returns true, if the polygon touches the other polygon.

The polygons touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

(4) Signature: *[const]* bool **touches?** (const DSimplePolygon simple_polygon)

Description: Returns true, if the polygon touches the other polygon.

The polygons touch if they overlap or their contours share at least one point.

This method was introduced in version 0.25.1.

transform

(1) Signature: DPolygon ptr **transform** (const DCplxTrans t)

Description: Transforms the polygon with a complex transformation (in-place)

t: The transformation to apply.

Transforms the polygon with the given complex transformation. Modifies self and returns self. An out-of-place version which does not modify self is transformed.

This method has been introduced in version 0.24.

(2) Signature: DPolygon ptr **transform** (const DTrans t)

Description: Transforms the polygon (in-place)

t: The transformation to apply.

Transforms the polygon with the given transformation. Modifies self and returns self. An out-of-place version which does not modify self is transformed.

This method has been introduced in version 0.24.

transformed

(1) Signature: *[const]* Polygon **transformed** (const VCplxTrans t)

Description: Transforms the polygon with the given complex transformation

t: The magnifying transformation to apply

Returns: The transformed polygon (in this case an integer coordinate polygon)

This method has been introduced in version 0.25.

(2) Signature: *[const]* DPolygon **transformed** (const DTrans t)

Description: Transforms the polygon

t: The transformation to apply.

Returns: The transformed polygon.

Transforms the polygon with the given transformation. Does not modify the polygon but returns the transformed polygon.

(3) Signature: *[const]* DPolygon **transformed** (const DCplxTrans t)

Description: Transforms the polygon with a complex transformation

t:	The transformation to apply.
Returns:	The transformed polygon.

Transforms the polygon with the given complex transformation. Does not modify the polygon but returns the transformed polygon.

With version 0.25, the original 'transformed_cplx' method is deprecated and 'transformed' takes both simple and complex transformations.

transformed_cplx

Signature: *[const]* DPolygon **transformed_cplx** (const DCplxTrans t)

Description: Transforms the polygon with a complex transformation

t:	The transformation to apply.
Returns:	The transformed polygon.

Use of this method is deprecated. Use transformed instead

Transforms the polygon with the given complex transformation. Does not modify the polygon but returns the transformed polygon.

With version 0.25, the original 'transformed_cplx' method is deprecated and 'transformed' takes both simple and complex transformations.

4.78. API reference - Class LayerMap

Notation used in Ruby API documentation

Module: db

Description: An object representing an arbitrary mapping of physical layers to logical layers

"Physical" layers are stream layers or other separated layers in a CAD file. "Logical" layers are the layers present in a Layout object. Logical layers are represented by an integer index while physical layers are given by a layer and datatype number or name. A logical layer is created automatically in the layout on reading if it does not exist yet.

The mapping describes an association of a set of physical layers to a set of logical ones, where multiple physical layers can be mapped to a single logical one, which effectively merges the layers.

For each logical layer, a target layer can be specified. A target layer is the layer/datatype/name combination as which the logical layer appears in the layout. By using a target layer different from the source layer renaming a layer can be achieved while loading a layout. Another use case for that feature is to assign layer names to GDS layer/datatype combinations which are numerical only.

LayerMap objects are used in two ways: as input for the reader (inside a LoadLayoutOptions class) and as output from the reader (i.e. Layout::read method). For layer map objects used as input, the layer indexes (logical layers) can be consecutive numbers. They do not need to correspond with real layer indexes from a layout object. When used as output, the layer map's logical layers correspond to the layer indexes inside the layout that the layer map was used upon.

This is a sample how to use the LayerMap object. It maps all datatypes of layers 1, 2 and 3 to datatype 0 and assigns the names 'ONE', 'TWO' and 'THREE' to these layout layers:

```
lm = RBA::LayerMap::new
lm.map("1/0-255 : ONE (1/0)")
lm.map("2/0-255 : TWO (2/0)")
lm.map("3/0-255 : THREE (3/0)")

# read the layout using the layer map
lo = RBA::LoadLayoutOptions::new
lo.layer_map.assign(lm)
ly = RBA::Layout::new
ly.read("input.gds", lo)
```

1:n mapping is supported: a physical layer can be mapped to multiple logical layers using 'mmap' instead of 'map'. When using this variant, mapping acts additive. The following example will map layer 1, datatypes 0 to 255 to logical layer 0, and layer 1, datatype 17 to logical layers 0 plus 1:

```
lm = RBA::LayerMap::new
lm.map("1/0-255", 0)   # (can be 'mmap' too)
lm.mmap("1/17", 1)
```

'unmapping' allows removing a mapping. This allows creating 'holes' in mapping ranges. The following example maps layer 1, datatypes 0 to 16 and 18 to 255 to logical layer 0:

```
lm = RBA::LayerMap::new
lm.map("1/0-255", 0)
lm.unmap("1/17")
```

The LayerMap class has been introduced in version 0.18. Target layer have been introduced in version 0.20. 1:n mapping and unmapping has been introduced in version 0.27.

Public constructors

new LayerMap ptr	**new**	Creates a new object of this class

Public methods

	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
	void	assign	(const LayerMap other)	Assigns another object to self
	void	clear		Clears the map
[const]	new LayerMap ptr	dup		Creates a copy of self
[const]	bool	is_mapped?	(const LayerInfo layer)	Check, if a given physical layer is mapped
[const]	unsigned int[]	logicals	(const LayerInfo layer)	Returns the logical layers for a given physical layer.n@param layer The physical layer specified with an LayerInfo object.
	void	map	(const LayerInfo phys_layer, unsigned int log_layer)	Maps a physical layer to a logical one
	void	map	(const LayerInfo phys_layer, unsigned int log_layer, const LayerInfo target_layer)	Maps a physical layer to a logical one with a target layer
	void	map	(const LayerInfo pl_start, const LayerInfo pl_stop, unsigned int log_layer)	Maps a physical layer interval to a logical one
	void	map	(const LayerInfo pl_start, const LayerInfo pl_stop, unsigned int log_layer, const LayerInfo layer_properties)	Maps a physical layer interval to a logical one with a target layer
	void	map	(string map_expr, int log_layer = -1)	Maps a physical layer given by a string to a logical one
[const]	LayerInfo	mapping	(unsigned int log_layer)	Returns the mapped physical (or target if one is specified) layer for a given logical layer
[const]	string	mapping_str	(unsigned int log_layer)	Returns the mapping string for a given logical layer

	void	**mmap**	(const LayerInfo phys_layer, unsigned int log_layer)	Maps a physical layer to a logical one and adds to existing mappings
	void	**mmap**	(const LayerInfo phys_layer, unsigned int log_layer, const LayerInfo target_layer)	Maps a physical layer to a logical one, adds to existing mappings and specifies a target layer
	void	**mmap**	(const LayerInfo pl_start, const LayerInfo pl_stop, unsigned int log_layer)	Maps a physical layer from the given interval to a logical one and adds to existing mappings
	void	**mmap**	(const LayerInfo pl_start, const LayerInfo pl_stop, unsigned int log_layer, const LayerInfo layer_properties)	Maps a physical layer from the given interval to a logical one, adds to existing mappings and specifies a target layer
	void	**mmap**	(string map_expr, int log_layer = -1)	Maps a physical layer given by an expression to a logical one and adds to existing mappings
[const]	string	**to_string**		Converts a layer mapping object to a string
	void	**unmap**	(const LayerInfo phys_layer)	Unmaps the given layer
	void	**unmap**	(const LayerInfo pl_start, const LayerInfo pl_stop)	Unmaps the layers from the given interval
	void	**unmap**	(string expr)	Unmaps the layers from the given expression

Public static methods and constants

LayerMap	**from_string**	(string s)	Creates a layer map from the given string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	int	**logical**	(const LayerInfo layer)	Use of this method is deprecated

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const LayerMap other)

Description: Assigns another object to self

clear

Signature: void **clear**

Description: Clears the map

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new LayerMap ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_string

Signature: *[static]* LayerMap **from_string** (string s)

Description: Creates a layer map from the given string

The format of the string is that used in layer mapping files: one mapping entry per line, comments are allowed using '#' or '//'. The format of each line is that used in the 'map(string, index)' method.

This method has been introduced in version 0.23.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_mapped?

Signature: *[const]* bool **is_mapped?** (const LayerInfo layer)

Description: Check, if a given physical layer is mapped

layer:	The physical layer specified with an LayerInfo object.
Returns:	True, if the layer is mapped.

logical

Signature: *[const]* int **logical** (const LayerInfo layer)

Description: Returns the logical layer (the layer index in the layout object) for a given physical layer.n@param layer The physical layer specified with an LayerInfo object.

Returns: The logical layer index or -1 if the layer is not mapped.

Use of this method is deprecated

This method is deprecated with version 0.27 as in this version, layers can be mapped to multiple targets which this method can't capture. Use logicals instead.

logicals

Signature: *[const]* unsigned int[] **logicals** (const LayerInfo layer)

Description: Returns the logical layers for a given physical layer.n@param layer The physical layer specified with an LayerInfo object.

Returns: This list of logical layers this physical layer as mapped to or empty if there is no mapping.

This method has been introduced in version 0.27.

map

(1) Signature: void **map** (const LayerInfo phys_layer, unsigned int log_layer)

Description: Maps a physical layer to a logical one

phys_layer: The physical layer (a LayerInfo object).

log_layer: The logical layer to which the physical layer is mapped.

In general, there may be more than one physical layer mapped to one logical layer. This method will add the given physical layer to the mapping for the logical layer.

(2) Signature: void **map** (const LayerInfo phys_layer, unsigned int log_layer, const LayerInfo target_layer)

Description: Maps a physical layer to a logical one with a target layer

phys_layer: The physical layer (a LayerInfo object).

log_layer: The logical layer to which the physical layer is mapped.

target_layer: The properties of the layer that will be created unless it already exists.

In general, there may be more than one physical layer mapped to one logical layer. This method will add the given physical layer to the mapping for the logical layer.

This method has been added in version 0.20.

(3) Signature: void **map** (const LayerInfo pl_start, const LayerInfo pl_stop, unsigned int log_layer)

Description: Maps a physical layer interval to a logical one

pl_start: The first physical layer (a LayerInfo object).

pl_stop: The last physical layer (a LayerInfo object).

log_layer: The logical layer to which the physical layers are mapped.

This method maps an interval of layers l1..l2 and datatypes d1..d2 to the mapping for the given logical layer. l1 and d1 are given by the pl_start argument, while l2 and d2 are given by the pl_stop argument.

(4) Signature: void **map** (const LayerInfo pl_start, const LayerInfo pl_stop, unsigned int log_layer, const LayerInfo layer_properties)

Description: Maps a physical layer interval to a logical one with a target layer

pl_start: The first physical layer (a LayerInfo object).

pl_stop: The last physical layer (a LayerInfo object).

log_layer: The logical layer to which the physical layers are mapped.

target_layer: The properties of the layer that will be created unless it already exists.

This method maps an interval of layers l1..l2 and datatypes d1..d2 to the mapping for the given logical layer. l1 and d1 are given by the pl_start argument, while l2 and d2 are given by the pl_stop argument. This method has been added in version 0.20.

(5) Signature: void **map** (string map_expr, int log_layer = -1)

Description: Maps a physical layer given by a string to a logical one

map_expr: The string describing the physical layer to map.

log_layer: The logical layer to which the physical layers are mapped.

The string expression is constructed using the syntax: "list[/list][;..]" for layer/datatype pairs. "list" is a sequence of numbers, separated by comma values or a range separated by a hyphen. Examples are: "1/2", "1-5/0", "1,2,5/0", "1/5;5/6".

layer/datatype wildcards can be specified with "*". When "*" is used for the upper limit, it is equivalent to "all layer above". When used alone, it is equivalent to "all layers". Examples: "1 / *", "* / 10-*"

Named layers are specified simply by specifying the name, if necessary in single or double quotes (if the name begins with a digit or contains non-word characters). layer/datatype and name descriptions can be mixed, i.e. "AA;1/5" (meaning: name "AA" or layer 1/datatype 5).

A target layer can be specified with the ":<target>" notation, where target is a valid string for a LayerProperties() object.

A target can include relative layer/datatype specifications and wildcards. For example, "1-10/0: *+1/0" will add 1 to the original layer number. "1-10/0-50: * / *" will use the original layers.

If the logical layer is negative or omitted, the method will select the next available one.

Target mapping has been added in version 0.20. The logical layer is optional since version 0.28.

mapping

Signature: *[const]* LayerInfo **mapping** (unsigned int log_layer)

Description: Returns the mapped physical (or target if one is specified) layer for a given logical layer

log_layer: The logical layer for which the mapping is requested.

Returns: A LayerInfo object which is the physical layer mapped to the logical layer.

In general, there may be more than one physical layer mapped to one logical layer. This method will return a single one of them. It will return the one with the lowest layer and datatype.

mapping_str

Signature: *[const]* string **mapping_str** (unsigned int log_layer)

Description: Returns the mapping string for a given logical layer

log_layer: The logical layer for which the mapping is requested.

Returns: A string describing the mapping.

The mapping string is compatible with the string that the "map" method accepts.

mmap

(1) Signature: void **mmap** (const LayerInfo phys_layer, unsigned int log_layer)

Description: Maps a physical layer to a logical one and adds to existing mappings

This method acts like the corresponding 'map' method, but adds the logical layer to the receivers of the given physical one. Hence this method implements 1:n mapping capabilities. For backward compatibility, 'map' still substitutes mapping.

Multi-mapping has been added in version 0.27.

(2) Signature: void **mmap** (const LayerInfo phys_layer, unsigned int log_layer, const LayerInfo target_layer)

Description: Maps a physical layer to a logical one, adds to existing mappings and specifies a target layer

This method acts like the corresponding 'map' method, but adds the logical layer to the receivers of the given physical one. Hence this method implements 1:n mapping capabilities. For backward compatibility, 'map' still substitutes mapping.

Multi-mapping has been added in version 0.27.

(3) Signature: void **mmap** (const LayerInfo pl_start, const LayerInfo pl_stop, unsigned int log_layer)

Description: Maps a physical layer from the given interval to a logical one and adds to existing mappings

This method acts like the corresponding 'map' method, but adds the logical layer to the receivers of the given physical one. Hence this method implements 1:n mapping capabilities. For backward compatibility, 'map' still substitutes mapping.

Multi-mapping has been added in version 0.27.

(4) Signature: void **mmap** (const LayerInfo pl_start, const LayerInfo pl_stop, unsigned int log_layer, const LayerInfo layer_properties)

Description: Maps a physical layer from the given interval to a logical one, adds to existing mappings and specifies a target layer

This method acts like the corresponding 'map' method, but adds the logical layer to the receivers of the given physical one. Hence this method implements 1:n mapping capabilities. For backward compatibility, 'map' still substitutes mapping.

Multi-mapping has been added in version 0.27.

(5) Signature: void **mmap** (string map_expr, int log_layer = -1)

Description: Maps a physical layer given by an expression to a logical one and adds to existing mappings

This method acts like the corresponding 'map' method, but adds the logical layer to the receivers of the given physical one. Hence this method implements 1:n mapping capabilities. For backward compatibility, 'map' still substitutes mapping.

If the logical layer is negative or omitted, the method will select the next available one.

Multi-mapping has been added in version 0.27. The logical layer is optional since version 0.28.

new

Signature: *[static]* new LayerMap ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

to_string

Signature: *[const]* string **to_string**

Description: Converts a layer mapping object to a string

This method is the inverse of the from_string method.

This method has been introduced in version 0.23.

unmap

(1) Signature: void **unmap** (const LayerInfo phys_layer)

Description: Unmaps the given layer

Unmapping will remove the specific layer from the mapping. This method allows generating 'mapping holes' by first mapping a range and then unmapping parts of it.

This method has been introduced in version 0.27.

(2) Signature: void **unmap** (const LayerInfo pl_start, const LayerInfo pl_stop)

Description: Unmaps the layers from the given interval

This method has been introduced in version 0.27.

(3) Signature: void **unmap** (string expr)

Description: Unmaps the layers from the given expression

This method has been introduced in version 0.27.

4.79. API reference - Class LoadLayoutOptions

Notation used in Ruby API documentation

Module: db

Description: Layout reader options

Sub-classes: CellConflictResolution

This object describes various layer reader options used for loading layouts.

This class has been introduced in version 0.18.

Public constructors

new LoadLayoutOptions ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const LoadLayoutOptior other)	Assigns another object to self
[const]	LoadLayoutOptions::CellConflictResolution	**cell_conflict_resolution**		Gets the cell conflict resolution mode
	void	**cell_conflict_resolution=**	(LoadLayoutOptio mode)	Sets the cell conflict resolution mode
	void	**cif_create_other_layers=**	(bool create)	Specifies whether other layers shall be created
[const]	bool	**cif_create_other_layers?**		Gets a value indicating whether other layers shall be created
[const]	double	**cif_dbu**		Specifies the database unit which the reader uses and produces
	void	**cif_dbu=**	(double dbu)	Specifies the database unit which the reader uses and produces
	void	**cif_keep_layer_names=**	(bool keep)	Gets a value indicating whether layer names are kept

[const]	bool	**cif_keep_layer_names?**		Gets a value indicating whether layer names are kept
	LayerMap	**cif_layer_map**		Gets the layer map
	void	**cif_layer_map=**	(const LayerMap map)	Sets the layer map
	void	**cif_select_all_layers**		Selects all layers and disables the layer map
	void	**cif_set_layer_map**	(const LayerMap map, bool create_other_laye	Sets the layer map
[const]	unsigned int	**cif_wire_mode**		Specifies how to read 'W' objects
	void	**cif_wire_mode=**	(unsigned int mode)	How to read 'W' objects
	void	**create_other_layers=**	(bool create)	Specifies whether other layers shall be created
[const]	bool	**create_other_layers?**		Gets a value indicating whether other layers shall be created
[const]	new LoadLayoutOptions ptr	**dup**		Creates a copy of self
[const]	double	**dxf_circle_accuracy**		Gets the accuracy of the circle approximation
	void	**dxf_circle_accuracy=**	(double accuracy)	Specifies the accuracy of the circle approximation
[const]	int	**dxf_circle_points**		Gets the number of points used per full circle for arc interpolation
	void	**dxf_circle_points=**	(int points)	Specifies the number of points used per full circle for arc interpolation
[const]	double	**dxf_contour_accuracy**		Gets the accuracy for contour closing
	void	**dxf_contour_accuracy=**	(double accuracy)	Specifies the accuracy for contour closing
	void	**dxf_create_other_layers=**	(bool create)	Specifies whether other layers shall be created
[const]	bool	**dxf_create_other_layers?**		Gets a value indicating whether other layers shall be created
[const]	double	**dxf_dbu**		Specifies the database unit which the reader uses and produces
	void	**dxf_dbu=**	(double dbu)	Specifies the database unit which the reader uses and produces

	void	**dxf_keep_layer_names=**	(bool keep)	Gets a value indicating whether layer names are kept
[const]	bool	**dxf_keep_layer_names?**		Gets a value indicating whether layer names are kept
	void	**dxf_keep_other_cells=**	(bool value)	If this option is set to true, all cells are kept, not only the top cell and its children
[const]	bool	**dxf_keep_other_cells?**		If this option is true, all cells are kept, not only the top cell and its children
	LayerMap	**dxf_layer_map**		Gets the layer map
	void	**dxf_layer_map=**	(const LayerMap map)	Sets the layer map
[const]	int	**dxf_polyline_mode**		Specifies whether closed POLYLINE and LWPOLYLINE entities with width 0 are converted to polygons.
	void	**dxf_polyline_mode=**	(int mode)	Specifies how to treat POLYLINE/ LWPOLYLINE entities.
	void	**dxf_render_texts_as_polyg**	(bool value)	If this option is set to true, text objects are rendered as polygons
[const]	bool	**dxf_render_texts_as_polygons?**		If this option is true, text objects are rendered as polygons
	void	**dxf_select_all_layers**		Selects all layers and disables the layer map
	void	**dxf_set_layer_map**	(const LayerMap map, bool create_other_layers)	Sets the layer map
[const]	double	**dxf_text_scaling**		Gets the text scaling factor (see dxf_text_scaling=)
	void	**dxf_text_scaling=**	(double unit)	Specifies the text scaling in percent of the default scaling
[const]	double	**dxf_unit**		Specifies the unit in which the DXF file is drawn
	void	**dxf_unit=**	(double unit)	Specifies the unit in which the DXF file is drawn.
	void	**gds2_allow_big_records=**	(bool flag)	Allows big records with more than 32767 bytes
[const]	bool	**gds2_allow_big_records?**		Gets a value specifying whether to allow big records with a length of 32768 to 65535 bytes.

	void	gds2_allow_multi_xy_recor	(bool flag)	Allows the use of multiple XY records in BOUNDARY elements for unlimited large polygons
[const]	bool	gds2_allow_multi_xy_records?		Gets a value specifying whether to allow big polygons with multiple XY records.
[const]	unsigned int	gds2_box_mode		Gets a value specifying how to treat BOX records
	void	gds2_box_mode=	(unsigned int mode)	Sets a value specifying how to treat BOX records
	LayerMap	layer_map		Gets the layer map
	void	layer_map=	(const LayerMap map)	Sets the layer map, but does not affect the "create_other_layers" flag.
	LEFDEFReaderCon	lefdef_config		Gets a copy of the LEF/DEF reader configuration
	void	lefdef_config=	(const LEFDEFReaderConfiguration config)	Sets the LEF/DEF reader configuration
	void	mag_create_other_layers=	(bool create)	Specifies whether other layers shall be created
[const]	bool	mag_create_other_layers?		Gets a value indicating whether other layers shall be created
[const]	double	mag_dbu		Specifies the database unit which the reader uses and produces
	void	mag_dbu=	(double dbu)	Specifies the database unit which the reader uses and produces
	void	mag_keep_layer_names=	(bool keep)	Gets a value indicating whether layer names are kept
[const]	bool	mag_keep_layer_names?		Gets a value indicating whether layer names are kept
[const]	double	mag_lambda		Gets the lambda value
	void	mag_lambda=	(double lambda)	Specifies the lambda value to used for reading
	LayerMap	mag_layer_map		Gets the layer map
	void	mag_layer_map=	(const LayerMap map)	Sets the layer map
[const]	string[]	mag_library_paths		Gets the locations where to look up libraries (in this order)

	void	**mag_library_paths=**	(string[] lib_paths)	Specifies the locations where to look up libraries (in this order)
	void	**mag_merge=**	(bool merge)	Sets a value indicating whether boxes are merged into polygons
[const]	bool	**mag_merge?**		Gets a value indicating whether boxes are merged into polygons
	void	**mag_select_all_layers**		Selects all layers and disables the layer map
	void	**mag_set_layer_map**	(const LayerMap map, bool create_other_layers)	Sets the layer map
	void	**properties_enabled=**	(bool enabled)	Specifies whether properties should be read
[const]	bool	**properties_enabled?**		Gets a value indicating whether properties shall be read
	void	**select_all_layers**		Selects all layers and disables the layer map
	void	**set_layer_map**	(const LayerMap map, bool create_other_layers)	Sets the layer map
	void	**text_enabled=**	(bool enabled)	Specifies whether text objects shall be read
[const]	bool	**text_enabled?**		Gets a value indicating whether text objects shall be read
[const]	int	**warn_level**		Sets the warning level.
	void	**warn_level=**	(int level)	Sets the warning level.

Public static methods and constants

[static,const]	LoadLayoutOptions::CellConflictRe	**AddToCell**		Add content to existing cell
[static,const]	LoadLayoutOptions::CellConflictResolution	**OverwriteCell**		The old cell is overwritten entirely (including child cells which are not used otherwise)
[static,const]	LoadLayoutOptions::CellConflictRe	**RenameCell**		The new cell will be renamed to become unique
[static,const]	LoadLayoutOptions::CellConflictResolution	**SkipNewCell**		The new cell is skipped entirely (including child cells which are not used otherwise)
	LoadLayoutOptions	**from_technology**	(string technology	Gets the reader options of a given technology

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**dxf_keep_other_cells**	Use of this method is deprecated. Use dxf_keep_other_cells? instead
[const]	bool	**dxf_render_texts_as_polygo**	Use of this method is deprecated. Use dxf_render_texts_as_polygons? instead
[const]	bool	**gds2_allow_big_records**	Use of this method is deprecated. Use gds2_allow_big_records? instead
[const]	bool	**gds2_allow_multi_xy_record**	Use of this method is deprecated. Use gds2_allow_multi_xy_records? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead
[const]	bool	**is_properties_enabled?**	Use of this method is deprecated. Use properties_enabled? instead
[const]	bool	**is_text_enabled?**	Use of this method is deprecated. Use text_enabled? instead

Detailed description

AddToCell

Signature: *[static,const]* LoadLayoutOptions::CellConflictResolution **AddToCell**

Description: Add content to existing cell

This is the mode use in before version 0.27. Content of new cells is simply added to existing cells with the same name.

Python specific notes:
The object exposes a readable attribute 'AddToCell'. This is the getter.

OverwriteCell

Signature: *[static,const]* LoadLayoutOptions::CellConflictResolution **OverwriteCell**

Description: The old cell is overwritten entirely (including child cells which are not used otherwise)

Python specific notes:
The object exposes a readable attribute 'OverwriteCell'. This is the getter.

RenameCell

Signature: *[static,const]* LoadLayoutOptions::CellConflictResolution **RenameCell**

Description: The new cell will be renamed to become unique

Python specific notes:
The object exposes a readable attribute 'RenameCell'. This is the getter.

SkipNewCell

Signature: *[static,const]* LoadLayoutOptions::CellConflictResolution **SkipNewCell**

Description: The new cell is skipped entirely (including child cells which are not used otherwise)

Python specific notes:
The object exposes a readable attribute 'SkipNewCell'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const LoadLayoutOptions other)

Description: Assigns another object to self

cell_conflict_resolution

Signature: *[const]* LoadLayoutOptions::CellConflictResolution **cell_conflict_resolution**

Description: Gets the cell conflict resolution mode

Multiple layout files can be collected into a single Layout object by reading file after file into the Layout object. Cells with same names are considered a conflict. This mode indicates how such conflicts are resolved. See LoadLayoutOptions::CellConflictResolution for the values allowed. The default mode is LoadLayoutOptions::CellConflictResolution#AddToCell.

This option has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'cell_conflict_resolution'. This is the getter.

cell_conflict_resolution=

Signature: void **cell_conflict_resolution=** (LoadLayoutOptions::CellConflictResolution mode)

Description: Sets the cell conflict resolution mode

See cell_conflict_resolution for details about this option.

This option has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'cell_conflict_resolution'. This is the setter.

cif_create_other_layers=

Signature: void **cif_create_other_layers=** (bool create)

Description: Specifies whether other layers shall be created

create: True, if other layers will be created.

See cif_create_other_layers? for a description of this attribute.

This method has been added in version 0.25 and replaces the respective global option in LoadLayoutOptions in a format-specific fashion.

Python specific notes:
The object exposes a writable attribute 'cif_create_other_layers'. This is the setter.

cif_create_other_layers?

Signature: *[const]* bool **cif_create_other_layers?**

Description: Gets a value indicating whether other layers shall be created

Returns: True, if other layers will be created.

This attribute acts together with a layer map (see cif_layer_map=). Layers not listed in this map are created as well when cif_create_other_layers? is true. Otherwise they are ignored.

This method has been added in version 0.25 and replaces the respective global option in LoadLayoutOptions in a format-specific fashion.

Python specific notes:
The object exposes a readable attribute 'cif_create_other_layers'. This is the getter.

cif_dbu

Signature: *[const]* double **cif_dbu**

Description: Specifies the database unit which the reader uses and produces

See cif_dbu= method for a description of this property. This property has been added in version 0.21.

Python specific notes:
The object exposes a readable attribute 'cif_dbu'. This is the getter.

cif_dbu=

Signature: void **cif_dbu=** (double dbu)

Description: Specifies the database unit which the reader uses and produces

This property has been added in version 0.21.

Python specific notes:
The object exposes a writable attribute 'cif_dbu'. This is the setter.

cif_keep_layer_names=

Signature: void **cif_keep_layer_names=** (bool keep)

Description: Gets a value indicating whether layer names are kept

keep: True, if layer names are to be kept.

See cif_keep_layer_names? for a description of this property.

This method has been added in version 0.25.3.

Python specific notes:
The object exposes a writable attribute 'cif_keep_layer_names'. This is the setter.

cif_keep_layer_names?

Signature: *[const]* bool **cif_keep_layer_names?**

Description: Gets a value indicating whether layer names are kept

Returns: True, if layer names are kept.

When set to true, no attempt is made to translate layer names to GDS layer/datatype numbers. If set to false (the default), a layer named "L2D15" will be translated to GDS layer 2, datatype 15.

This method has been added in version 0.25.3.

Python specific notes:
The object exposes a readable attribute 'cif_keep_layer_names'. This is the getter.

cif_layer_map

Signature: LayerMap **cif_layer_map**

Description: Gets the layer map

Returns: A reference to the layer map

This method has been added in version 0.25 and replaces the respective global option in LoadLayoutOptions in a format-specific fashion.

Python note: this method has been turned into a property in version 0.26.

Python specific notes:
The object exposes a readable attribute 'cif_layer_map'. This is the getter.

cif_layer_map=

Signature: void **cif_layer_map=** (const LayerMap map)

Description: Sets the layer map

map: The layer map to set.

This sets a layer mapping for the reader. Unlike cif_set_layer_map, the 'create_other_layers' flag is not changed.

This convenience method has been added in version 0.26.

Python specific notes:
The object exposes a writable attribute 'cif_layer_map'. This is the setter.

cif_select_all_layers

Signature: void **cif_select_all_layers**

Description: Selects all layers and disables the layer map

This disables any layer map and enables reading of all layers. New layers will be created when required.

This method has been added in version 0.25 and replaces the respective global option in LoadLayoutOptions in a format-specific fashion.

cif_set_layer_map

Signature: void **cif_set_layer_map** (const LayerMap map, bool create_other_layers)

Description: Sets the layer map

map:	The layer map to set.
create_other_layers:	The flag indicating whether other layers will be created as well. Set to false to read only the layers in the layer map.

This sets a layer mapping for the reader. The layer map allows selection and translation of the original layers, for example to assign layer/datatype numbers to the named layers.

This method has been added in version 0.25 and replaces the respective global option in LoadLayoutOptions in a format-specific fashion.

cif_wire_mode

Signature: *[const]* unsigned int **cif_wire_mode**

Description: Specifies how to read 'W' objects

See cif_wire_mode= method for a description of this mode. This property has been added in version 0.21 and was renamed to cif_wire_mode in 0.25.

Python specific notes:
The object exposes a readable attribute 'cif_wire_mode'. This is the getter.

cif_wire_mode=

Signature: void **cif_wire_mode=** (unsigned int mode)

Description: How to read 'W' objects

This property specifies how to read 'W' (wire) objects. Allowed values are 0 (as square ended paths), 1 (as flush ended paths), 2 (as round paths)

This property has been added in version 0.21.

Python specific notes:
The object exposes a writable attribute 'cif_wire_mode'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

create_other_layers=

Signature: void **create_other_layers=** (bool create)

Description: Specifies whether other layers shall be created

create:	True, if other layers should be created.

See create_other_layers? for a description of this attribute.

Starting with version 0.25 this option only applies to GDS2 and OASIS format. Other formats provide their own configuration.

Python specific notes:
The object exposes a writable attribute 'create_other_layers'. This is the setter.

create_other_layers?

Signature: *[const]* bool **create_other_layers?**

Description: Gets a value indicating whether other layers shall be created

Returns:	True, if other layers should be created.

This attribute acts together with a layer map (see layer_map=). Layers not listed in this map are created as well when create_other_layers? is true. Otherwise they are ignored.

Starting with version 0.25 this option only applies to GDS2 and OASIS format. Other formats provide their own configuration.

Python specific notes:
The object exposes a readable attribute 'create_other_layers'. This is the getter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new LoadLayoutOptions ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

dxf_circle_accuracy

Signature: *[const]* double **dxf_circle_accuracy**

Description: Gets the accuracy of the circle approximation

This property has been added in version 0.24.9.

Python specific notes:
The object exposes a readable attribute 'dxf_circle_accuracy'. This is the getter.

dxf_circle_accuracy=

Signature: void **dxf_circle_accuracy=** (double accuracy)

Description: Specifies the accuracy of the circle approximation

In addition to the number of points per circle, the circle accuracy can be specified. If set to a value larger than the database unit, the number of points per circle will be chosen such that the deviation from the ideal circle becomes less than this value.

The actual number of points will not become bigger than the points specified through dxf_circle_points=. The accuracy value is given in the DXF file units (see dxf_unit) which is usually micrometers.

dxf_circle_points and dxf_circle_accuracy also apply to other "round" structures such as arcs, ellipses and splines in the same sense than for circles.

This property has been added in version 0.24.9.

Python specific notes:
The object exposes a writable attribute 'dxf_circle_accuracy'. This is the setter.

dxf_circle_points

Signature: *[const]* int **dxf_circle_points**

Description: Gets the number of points used per full circle for arc interpolation

This property has been added in version 0.21.6.

Python specific notes:

The object exposes a readable attribute 'dxf_circle_points'. This is the getter.

dxf_circle_points=

Signature: void **dxf_circle_points=** (int points)

Description: Specifies the number of points used per full circle for arc interpolation

See also dxf_circle_accuracy for how to specify the number of points based on an approximation accuracy.

dxf_circle_points and dxf_circle_accuracy also apply to other "round" structures such as arcs, ellipses and splines in the same sense than for circles.

This property has been added in version 0.21.6.

Python specific notes:
The object exposes a writable attribute 'dxf_circle_points'. This is the setter.

dxf_contour_accuracy

Signature: *[const]* double **dxf_contour_accuracy**

Description: Gets the accuracy for contour closing

This property has been added in version 0.25.3.

Python specific notes:
The object exposes a readable attribute 'dxf_contour_accuracy'. This is the getter.

dxf_contour_accuracy=

Signature: void **dxf_contour_accuracy=** (double accuracy)

Description: Specifies the accuracy for contour closing

When polylines need to be connected or closed, this value is used to indicate the accuracy. This is the value (in DXF units) by which points may be separated and still be considered connected. The default is 0.0 which implies exact (within one DBU) closing.

This value is effective in polyline mode 3 and 4.

This property has been added in version 0.25.3.

Python specific notes:
The object exposes a writable attribute 'dxf_contour_accuracy'. This is the setter.

dxf_create_other_layers=

Signature: void **dxf_create_other_layers=** (bool create)

Description: Specifies whether other layers shall be created

create: True, if other layers will be created.

See dxf_create_other_layers? for a description of this attribute.

This method has been added in version 0.25 and replaces the respective global option in LoadLayoutOptions in a format-specific fashion.

Python specific notes:
The object exposes a writable attribute 'dxf_create_other_layers'. This is the setter.

dxf_create_other_layers?

Signature: *[const]* bool **dxf_create_other_layers?**

Description: Gets a value indicating whether other layers shall be created

Returns: True, if other layers will be created.

This attribute acts together with a layer map (see dxf_layer_map=). Layers not listed in this map are created as well when dxf_create_other_layers? is true. Otherwise they are ignored.

This method has been added in version 0.25 and replaces the respective global option in LoadLayoutOptions in a format-specific fashion.

Python specific notes:

The object exposes a readable attribute 'dxf_create_other_layers'. This is the getter.

dxf_dbu

Signature: *[const]* double **dxf_dbu**

Description: Specifies the database unit which the reader uses and produces

This property has been added in version 0.21.

Python specific notes:
The object exposes a readable attribute 'dxf_dbu'. This is the getter.

dxf_dbu=

Signature: void **dxf_dbu=** (double dbu)

Description: Specifies the database unit which the reader uses and produces

This property has been added in version 0.21.

Python specific notes:
The object exposes a writable attribute 'dxf_dbu'. This is the setter.

dxf_keep_layer_names=

Signature: void **dxf_keep_layer_names=** (bool keep)

Description: Gets a value indicating whether layer names are kept

keep: True, if layer names are to be kept.

See cif_keep_layer_names? for a description of this property.

This method has been added in version 0.25.3.

Python specific notes:
The object exposes a writable attribute 'dxf_keep_layer_names'. This is the setter.

dxf_keep_layer_names?

Signature: *[const]* bool **dxf_keep_layer_names?**

Description: Gets a value indicating whether layer names are kept

Returns: True, if layer names are kept.

When set to true, no attempt is made to translate layer names to GDS layer/datatype numbers. If set to false (the default), a layer named "L2D15" will be translated to GDS layer 2, datatype 15.

This method has been added in version 0.25.3.

Python specific notes:
The object exposes a readable attribute 'dxf_keep_layer_names'. This is the getter.

dxf_keep_other_cells

Signature: *[const]* bool **dxf_keep_other_cells**

Description: If this option is true, all cells are kept, not only the top cell and its children

Use of this method is deprecated. Use dxf_keep_other_cells? instead

This property has been added in version 0.21.15.

Python specific notes:
The object exposes a readable attribute 'dxf_keep_other_cells'. This is the getter.
This method is available as 'dxf_keep_other_cells_' in Python to distiguish it from the property with the same name.

dxf_keep_other_cells=

Signature: void **dxf_keep_other_cells=** (bool value)

Description: If this option is set to true, all cells are kept, not only the top cell and its children

This property has been added in version 0.21.15.

Python specific notes:

The object exposes a writable attribute 'dxf_keep_other_cells'. This is the setter.

dxf_keep_other_cells?

Signature: *[const]* bool **dxf_keep_other_cells?**

Description: If this option is true, all cells are kept, not only the top cell and its children

This property has been added in version 0.21.15.

Python specific notes:
The object exposes a readable attribute 'dxf_keep_other_cells'. This is the getter.
This method is available as 'dxf_keep_other_cells_' in Python to distiguish it from the property with the same name.

dxf_layer_map

Signature: LayerMap **dxf_layer_map**

Description: Gets the layer map

Returns: A reference to the layer map

This method has been added in version 0.25 and replaces the respective global option in LoadLayoutOptions in a format-specific fashion. Python note: this method has been turned into a property in version 0.26.

Python specific notes:
The object exposes a readable attribute 'dxf_layer_map'. This is the getter.

dxf_layer_map=

Signature: void **dxf_layer_map=** (const LayerMap map)

Description: Sets the layer map

map: The layer map to set.

This sets a layer mapping for the reader. Unlike dxf_set_layer_map, the 'create_other_layers' flag is not changed.

This convenience method has been added in version 0.26.

Python specific notes:
The object exposes a writable attribute 'dxf_layer_map'. This is the setter.

dxf_polyline_mode

Signature: *[const]* int **dxf_polyline_mode**

Description: Specifies whether closed POLYLINE and LWPOLYLINE entities with width 0 are converted to polygons.

See dxf_polyline_mode= for a description of this property.

This property has been added in version 0.21.3.

Python specific notes:
The object exposes a readable attribute 'dxf_polyline_mode'. This is the getter.

dxf_polyline_mode=

Signature: void **dxf_polyline_mode=** (int mode)

Description: Specifies how to treat POLYLINE/LWPOLYLINE entities.

The mode is 0 (automatic), 1 (keep lines), 2 (create polygons from closed polylines with width = 0), 3 (merge all lines with width = 0 into polygons), 4 (as 3 plus auto-close open contours).

This property has been added in version 0.21.3.

Python specific notes:
The object exposes a writable attribute 'dxf_polyline_mode'. This is the setter.

dxf_render_texts_as_polygor

Signature: *[const]* bool **dxf_render_texts_as_polygons**

Description: If this option is true, text objects are rendered as polygons

Use of this method is deprecated. Use dxf_render_texts_as_polygons? instead

This property has been added in version 0.21.15.

Python specific notes:
The object exposes a readable attribute 'dxf_render_texts_as_polygons'. This is the getter.
This method is available as 'dxf_render_texts_as_polygons_' in Python to distiguish it from the property with the same name.

dxf_render_texts_as_polygons=

Signature: void **dxf_render_texts_as_polygons=** (bool value)

Description: If this option is set to true, text objects are rendered as polygons

This property has been added in version 0.21.15.

Python specific notes:
The object exposes a writable attribute 'dxf_render_texts_as_polygons'. This is the setter.

dxf_render_texts_as_polygon

Signature: *[const]* bool **dxf_render_texts_as_polygons?**

Description: If this option is true, text objects are rendered as polygons

This property has been added in version 0.21.15.

Python specific notes:
The object exposes a readable attribute 'dxf_render_texts_as_polygons'. This is the getter.
This method is available as 'dxf_render_texts_as_polygons_' in Python to distiguish it from the property with the same name.

dxf_select_all_layers

Signature: void **dxf_select_all_layers**

Description: Selects all layers and disables the layer map

This disables any layer map and enables reading of all layers. New layers will be created when required.

This method has been added in version 0.25 and replaces the respective global option in LoadLayoutOptions in a format-specific fashion.

dxf_set_layer_map

Signature: void **dxf_set_layer_map** (const LayerMap map, bool create_other_layers)

Description: Sets the layer map

map:	The layer map to set.
create_other_layers:	The flag indicating whether other layers will be created as well. Set to false to read only the layers in the layer map.

This sets a layer mapping for the reader. The layer map allows selection and translation of the original layers, for example to assign layer/datatype numbers to the named layers.

This method has been added in version 0.25 and replaces the respective global option in LoadLayoutOptions in a format-specific fashion.

dxf_text_scaling

Signature: *[const]* double **dxf_text_scaling**

Description: Gets the text scaling factor (see dxf_text_scaling=)

This property has been added in version 0.21.20.

Python specific notes:
The object exposes a readable attribute 'dxf_text_scaling'. This is the getter.

dxf_text_scaling=

Signature: void **dxf_text_scaling=** (double unit)

Description: Specifies the text scaling in percent of the default scaling

The default value 100, meaning that the letter pitch is roughly 92 percent of the specified text height. Decrease this value to get smaller fonts and increase it to get larger fonts.

This property has been added in version 0.21.20.

Python specific notes:
The object exposes a writable attribute 'dxf_text_scaling'. This is the setter.

dxf_unit

Signature: *[const]* double **dxf_unit**

Description: Specifies the unit in which the DXF file is drawn

This property has been added in version 0.21.3.

Python specific notes:
The object exposes a readable attribute 'dxf_unit'. This is the getter.

dxf_unit=

Signature: void **dxf_unit=** (double unit)

Description: Specifies the unit in which the DXF file is drawn.

This property has been added in version 0.21.3.

Python specific notes:
The object exposes a writable attribute 'dxf_unit'. This is the setter.

from_technology

Signature: *[static]* LoadLayoutOptions **from_technology** (string technology)

Description: Gets the reader options of a given technology

technology: The name of the technology to apply

Returns the reader options of a specific technology. If the technology name is not valid or an empty string, the reader options of the default technology are returned.

This method has been introduced in version 0.25

gds2_allow_big_records

Signature: *[const]* bool **gds2_allow_big_records**

Description: Gets a value specifying whether to allow big records with a length of 32768 to 65535 bytes.

Use of this method is deprecated. Use gds2_allow_big_records? instead

See gds2_allow_big_records= method for a description of this property. This property has been added in version 0.18.

Python specific notes:
The object exposes a readable attribute 'gds2_allow_big_records'. This is the getter.
This method is available as 'gds2_allow_big_records_' in Python to distiguish it from the property with the same name.

gds2_allow_big_records=

Signature: void **gds2_allow_big_records=** (bool flag)

Description: Allows big records with more than 32767 bytes

Setting this property to true allows larger records by treating the record length as unsigned short, which for example allows larger polygons (~8000 points rather than ~4000 points) without using multiple XY records. For strict compatibility with the standard, this property should be set to false. The default is true.

This property has been added in version 0.18.

Python specific notes:
The object exposes a writable attribute 'gds2_allow_big_records'. This is the setter.

gds2_allow_big_records?

Signature: *[const]* bool **gds2_allow_big_records?**

Description: Gets a value specifying whether to allow big records with a length of 32768 to 65535 bytes.

See gds2_allow_big_records= method for a description of this property. This property has been added in version 0.18.

Python specific notes:
The object exposes a readable attribute 'gds2_allow_big_records'. This is the getter.
This method is available as 'gds2_allow_big_records_' in Python to distiguish it from the property with the same name.

gds2_allow_multi_xy_records

Signature: *[const]* bool **gds2_allow_multi_xy_records**

Description: Gets a value specifying whether to allow big polygons with multiple XY records.

Use of this method is deprecated. Use gds2_allow_multi_xy_records? instead

See gds2_allow_multi_xy_records= method for a description of this property. This property has been added in version 0.18.

Python specific notes:
The object exposes a readable attribute 'gds2_allow_multi_xy_records'. This is the getter.
This method is available as 'gds2_allow_multi_xy_records_' in Python to distiguish it from the property with the same name.

gds2_allow_multi_xy_record

Signature: void **gds2_allow_multi_xy_records=** (bool flag)

Description: Allows the use of multiple XY records in BOUNDARY elements for unlimited large polygons

Setting this property to true allows big polygons that span over multiple XY records. For strict compatibility with the standard, this property should be set to false. The default is true.

This property has been added in version 0.18.

Python specific notes:
The object exposes a writable attribute 'gds2_allow_multi_xy_records'. This is the setter.

gds2_allow_multi_xy_records?

Signature: *[const]* bool **gds2_allow_multi_xy_records?**

Description: Gets a value specifying whether to allow big polygons with multiple XY records.

See gds2_allow_multi_xy_records= method for a description of this property. This property has been added in version 0.18.

Python specific notes:
The object exposes a readable attribute 'gds2_allow_multi_xy_records'. This is the getter.
This method is available as 'gds2_allow_multi_xy_records_' in Python to distiguish it from the property with the same name.

gds2_box_mode

Signature: *[const]* unsigned int **gds2_box_mode**

Description: Gets a value specifying how to treat BOX records

See gds2_box_mode= method for a description of this mode. This property has been added in version 0.18.

Python specific notes:
The object exposes a readable attribute 'gds2_box_mode'. This is the getter.

gds2_box_mode=

Signature: void **gds2_box_mode=** (unsigned int mode)

Description: Sets a value specifying how to treat BOX records

This property specifies how BOX records are treated. Allowed values are 0 (ignore), 1 (treat as rectangles), 2 (treat as boundaries) or 3 (treat as errors). The default is 1.

This property has been added in version 0.18.

Python specific notes:
The object exposes a writable attribute 'gds2_box_mode'. This is the setter.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_properties_enabled?

Signature: *[const]* bool **is_properties_enabled?**

Description: Gets a value indicating whether properties shall be read

Returns: True, if properties should be read.

Use of this method is deprecated. Use properties_enabled? instead

Starting with version 0.25 this option only applies to GDS2 and OASIS format. Other formats provide their own configuration.

Python specific notes:
The object exposes a readable attribute 'properties_enabled'. This is the getter.

is_text_enabled?

Signature: *[const]* bool **is_text_enabled?**

Description: Gets a value indicating whether text objects shall be read

Returns: True, if text objects should be read.

Use of this method is deprecated. Use text_enabled? instead

Starting with version 0.25 this option only applies to GDS2 and OASIS format. Other formats provide their own configuration.

Python specific notes:
The object exposes a readable attribute 'text_enabled'. This is the getter.

layer_map

Signature: LayerMap **layer_map**

Description: Gets the layer map

Returns: A reference to the layer map

Starting with version 0.25 this option only applies to GDS2 and OASIS format. Other formats provide their own configuration. Python note: this method has been turned into a property in version 0.26.

Python specific notes:
The object exposes a readable attribute 'layer_map'. This is the getter.

layer_map=

Signature: void **layer_map=** (const LayerMap map)

Description: Sets the layer map, but does not affect the "create_other_layers" flag.

map: The layer map to set.

Use create_other_layers? to enable or disable other layers not listed in the layer map.

This convenience method has been introduced with version 0.26.

Python specific notes:

The object exposes a writable attribute 'layer_map'. This is the setter.

lefdef_config

Signature: LEFDEFReaderConfiguration **lefdef_config**

Description: Gets a copy of the LEF/DEF reader configuration

The LEF/DEF reader configuration is wrapped in a separate object of class LEFDEFReaderConfiguration. See there for details. This method will return a copy of the reader configuration. To modify the configuration, modify the copy and set the modified configuration with lefdef_config=.

This method has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'lefdef_config'. This is the getter.

lefdef_config=

Signature: void **lefdef_config=** (const LEFDEFReaderConfiguration config)

Description: Sets the LEF/DEF reader configuration

This method has been added in version 0.25.

Python specific notes:
The object exposes a writable attribute 'lefdef_config'. This is the setter.

mag_create_other_layers=

Signature: void **mag_create_other_layers=** (bool create)

Description: Specifies whether other layers shall be created

create: True, if other layers will be created.

See mag_create_other_layers? for a description of this attribute.

This method has been added in version 0.26.2.

Python specific notes:
The object exposes a writable attribute 'mag_create_other_layers'. This is the setter.

mag_create_other_layers?

Signature: *[const]* bool **mag_create_other_layers?**

Description: Gets a value indicating whether other layers shall be created

Returns: True, if other layers will be created.

This attribute acts together with a layer map (see mag_layer_map=). Layers not listed in this map are created as well when mag_create_other_layers? is true. Otherwise they are ignored.

This method has been added in version 0.26.2.

Python specific notes:
The object exposes a readable attribute 'mag_create_other_layers'. This is the getter.

mag_dbu

Signature: *[const]* double **mag_dbu**

Description: Specifies the database unit which the reader uses and produces

See mag_dbu= method for a description of this property.

This property has been added in version 0.26.2.

Python specific notes:
The object exposes a readable attribute 'mag_dbu'. This is the getter.

mag_dbu=

Signature: void **mag_dbu=** (double dbu)

Description: Specifies the database unit which the reader uses and produces

The database unit is the final resolution of the produced layout. This physical resolution is usually defined by the layout system - GDS for example typically uses 1nm (mag_dbu=0.001). All

geometry in the MAG file will first be scaled to mag_lambda and is then brought to the database unit.

This property has been added in version 0.26.2.

Python specific notes:
The object exposes a writable attribute 'mag_dbu'. This is the setter.

mag_keep_layer_names=

Signature: void **mag_keep_layer_names=** (bool keep)

Description: Gets a value indicating whether layer names are kept

keep: True, if layer names are to be kept.

See mag_keep_layer_names? for a description of this property.

This method has been added in version 0.26.2.

Python specific notes:
The object exposes a writable attribute 'mag_keep_layer_names'. This is the setter.

mag_keep_layer_names?

Signature: *[const]* bool **mag_keep_layer_names?**

Description: Gets a value indicating whether layer names are kept

Returns: True, if layer names are kept.

When set to true, no attempt is made to translate layer names to GDS layer/datatype numbers. If set to false (the default), a layer named "L2D15" will be translated to GDS layer 2, datatype 15.

This method has been added in version 0.26.2.

Python specific notes:
The object exposes a readable attribute 'mag_keep_layer_names'. This is the getter.

mag_lambda

Signature: *[const]* double **mag_lambda**

Description: Gets the lambda value

See mag_lambda= method for a description of this attribute.

This property has been added in version 0.26.2.

Python specific notes:
The object exposes a readable attribute 'mag_lambda'. This is the getter.

mag_lambda=

Signature: void **mag_lambda=** (double lambda)

Description: Specifies the lambda value to used for reading

The lambda value is the basic unit of the layout. Magic draws layout as multiples of this basic unit. The layout read by the MAG reader will use the database unit specified by mag_dbu, but the physical layout coordinates will be multiples of mag_lambda.

This property has been added in version 0.26.2.

Python specific notes:
The object exposes a writable attribute 'mag_lambda'. This is the setter.

mag_layer_map

Signature: LayerMap **mag_layer_map**

Description: Gets the layer map

Returns: A reference to the layer map

This method has been added in version 0.26.2.

Python specific notes:
The object exposes a readable attribute 'mag_layer_map'. This is the getter.

mag_layer_map=

Signature: void **mag_layer_map=** (const LayerMap map)

Description: Sets the layer map

map: The layer map to set.

This sets a layer mapping for the reader. Unlike mag_set_layer_map, the 'create_other_layers' flag is not changed.

This method has been added in version 0.26.2.

Python specific notes:
The object exposes a writable attribute 'mag_layer_map'. This is the setter.

mag_library_paths

Signature: *[const]* string[] **mag_library_paths**

Description: Gets the locations where to look up libraries (in this order)

See mag_library_paths= method for a description of this attribute.

This property has been added in version 0.26.2.

Python specific notes:
The object exposes a readable attribute 'mag_library_paths'. This is the getter.

mag_library_paths=

Signature: void **mag_library_paths=** (string[] lib_paths)

Description: Specifies the locations where to look up libraries (in this order)

The reader will look up library reference in these paths when it can't find them locally. Relative paths in this collection are resolved relative to the initial file's path. Expression interpolation is supported in the path strings.

This property has been added in version 0.26.2.

Python specific notes:
The object exposes a writable attribute 'mag_library_paths'. This is the setter.

mag_merge=

Signature: void **mag_merge=** (bool merge)

Description: Sets a value indicating whether boxes are merged into polygons

merge: True, if boxes and triangles will be merged into polygons.

See mag_merge? for a description of this property.

This method has been added in version 0.26.2.

Python specific notes:
The object exposes a writable attribute 'mag_merge'. This is the setter.

mag_merge?

Signature: *[const]* bool **mag_merge?**

Description: Gets a value indicating whether boxes are merged into polygons

Returns: True, if boxes are merged.

When set to true, the boxes and triangles of the Magic layout files are merged into polygons where possible.

This method has been added in version 0.26.2.

Python specific notes:
The object exposes a readable attribute 'mag_merge'. This is the getter.

mag_select_all_layers

Signature: void **mag_select_all_layers**

Description: Selects all layers and disables the layer map

This disables any layer map and enables reading of all layers. New layers will be created when required.

This method has been added in version 0.26.2.

mag_set_layer_map

Signature: void **mag_set_layer_map** (const LayerMap map, bool create_other_layers)

Description: Sets the layer map

map:	The layer map to set.
create_other_layers:	The flag indicating whether other layers will be created as well. Set to false to read only the layers in the layer map.

This sets a layer mapping for the reader. The layer map allows selection and translation of the original layers, for example to assign layer/datatype numbers to the named layers.

This method has been added in version 0.26.2.

new

Signature: *[static]* new LoadLayoutOptions ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

properties_enabled=

Signature: void **properties_enabled=** (bool enabled)

Description: Specifies whether properties should be read

enabled:	True, if properties should be read.

Starting with version 0.25 this option only applies to GDS2 and OASIS format. Other formats provide their own configuration.

Python specific notes:
The object exposes a writable attribute 'properties_enabled'. This is the setter.

properties_enabled?

Signature: *[const]* bool **properties_enabled?**

Description: Gets a value indicating whether properties shall be read

Returns:	True, if properties should be read.

Starting with version 0.25 this option only applies to GDS2 and OASIS format. Other formats provide their own configuration.

Python specific notes:
The object exposes a readable attribute 'properties_enabled'. This is the getter.

select_all_layers

Signature: void **select_all_layers**

Description: Selects all layers and disables the layer map

This disables any layer map and enables reading of all layers. New layers will be created when required.

Starting with version 0.25 this method only applies to GDS2 and OASIS format. Other formats provide their own configuration.

set_layer_map

Signature: void **set_layer_map** (const LayerMap map, bool create_other_layers)

Description: Sets the layer map

map:	The layer map to set.@param create_other_layers The flag telling whether other layer should be created as well. Set to false if just the layers in the mapping table should be read.

This sets a layer mapping for the reader. The layer map allows selection and translation of the original layers, for example to add a layer name.

Starting with version 0.25 this option only applies to GDS2 and OASIS format. Other formats provide their own configuration.

text_enabled=

Signature: void **text_enabled=** (bool enabled)

Description: Specifies whether text objects shall be read

enabled: True, if text objects should be read.

Starting with version 0.25 this option only applies to GDS2 and OASIS format. Other formats provide their own configuration.

Python specific notes:
The object exposes a writable attribute 'text_enabled'. This is the setter.

text_enabled?

Signature: *[const]* bool **text_enabled?**

Description: Gets a value indicating whether text objects shall be read

Returns: True, if text objects should be read.

Starting with version 0.25 this option only applies to GDS2 and OASIS format. Other formats provide their own configuration.

Python specific notes:
The object exposes a readable attribute 'text_enabled'. This is the getter.

warn_level

Signature: *[const]* int **warn_level**

Description: Sets the warning level.

See warn_level= for details about this attribute.

This attribute has been added in version 0.28.

Python specific notes:
The object exposes a readable attribute 'warn_level'. This is the getter.

warn_level=

Signature: void **warn_level=** (int level)

Description: Sets the warning level.

The warning level is a reader-specific setting which enables or disables warnings on specific levels. Level 0 is always "warnings off". The default level is 1 which means "reasonable warnings emitted".

This attribute has been added in version 0.28.

Python specific notes:
The object exposes a writable attribute 'warn_level'. This is the setter.

4.80. API reference - Class LoadLayoutOptions::CellConflictResolution

Notation used in Ruby API documentation

Module: db

Description: This enum specifies how cell conflicts are handled if a layout read into another layout and a cell name conflict arises.

This class is equivalent to the class LoadLayoutOptions::CellConflictResolution

Until version 0.26.8 and before, the mode was always 'AddToCell'. On reading, a cell was 'reopened' when encountering a cell name which already existed. This mode is still the default. The other modes are made available to support other ways of merging layouts.

Proxy cells are never modified in the existing layout. Proxy cells are always local to their layout file. So if the existing cell is a proxy cell, the new cell will be renamed.

If the new or existing cell is a ghost cell, both cells are merged always.

This enum was introduced in version 0.27.

Public constructors

new LoadLayoutOptions::CellConflictResolution ptr	**new**	(int i)	Creates an enum from an integer value
new LoadLayoutOptions::CellConflictResolution ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const LoadLayoutOptions::CellC other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const LoadLayoutOptions::CellC other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const LoadLayoutOptions::CellC other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	LoadLayoutOptions::CellConflictRes	**AddToCell**	Add content to existing cell
[static,const]	LoadLayoutOptions::CellConflictResolution	**OverwriteCell**	The old cell is overwritten entirely (including child cells which are not used otherwise)
[static,const]	LoadLayoutOptions::CellConflictRes	**RenameCell**	The new cell will be renamed to become unique
[static,const]	LoadLayoutOptions::CellConflictResolution	**SkipNewCell**	The new cell is skipped entirely (including child cells which are not used otherwise)

Detailed description

!=

(1) Signature: *[const]* bool **!=** (const LoadLayoutOptions::CellConflictResolution other)
Description: Compares two enums for inequality

(2) Signature: *[const]* bool **!=** (int other)
Description: Compares an enum with an integer for inequality

<

(1) Signature: *[const]* bool **<** (const LoadLayoutOptions::CellConflictResolution other)
Description: Returns true if the first enum is less (in the enum symbol order) than the second

(2) Signature: *[const]* bool **<** (int other)
Description: Returns true if the enum is less (in the enum symbol order) than the integer value

==

(1) Signature: *[const]* bool **==** (const LoadLayoutOptions::CellConflictResolution other)
Description: Compares two enums

(2) Signature: *[const]* bool **==** (int other)
Description: Compares an enum with an integer value

AddToCell

Signature: *[static,const]* LoadLayoutOptions::CellConflictResolution **AddToCell**
Description: Add content to existing cell
This is the mode use in before version 0.27. Content of new cells is simply added to existing cells with the same name.
Python specific notes:
The object exposes a readable attribute 'AddToCell'. This is the getter.

OverwriteCell

Signature: *[static,const]* LoadLayoutOptions::CellConflictResolution **OverwriteCell**
Description: The old cell is overwritten entirely (including child cells which are not used otherwise)
Python specific notes:
The object exposes a readable attribute 'OverwriteCell'. This is the getter.

RenameCell

Signature: *[static,const]* LoadLayoutOptions::CellConflictResolution **RenameCell**

Description: The new cell will be renamed to become unique

Python specific notes:
The object exposes a readable attribute 'RenameCell'. This is the getter.

SkipNewCell

Signature: *[static,const]* LoadLayoutOptions::CellConflictResolution **SkipNewCell**

Description: The new cell is skipped entirely (including child cells which are not used otherwise)

Python specific notes:
The object exposes a readable attribute 'SkipNewCell'. This is the getter.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

new

(1) Signature: *[static]* new LoadLayoutOptions::CellConflictResolution ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new LoadLayoutOptions::CellConflictResolution ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.81. API reference - Class RecursiveInstanceIterator

Notation used in Ruby API documentation

Module: db

Description: An iterator delivering instances recursively

The iterator can be obtained from a cell and optionally a region. It simplifies retrieval of instances while considering subcells as well. Some options can be specified in addition, i.e. the hierarchy level to which to look into. The search can be confined to instances of certain cells (see targets=) or to certain regions. Subtrees can be selected for traversal or excluded from it (see select_cells).

This is some sample code:

```
# prints the effective instances of cell "A" as seen from the initial cell "cell"
iter = cell.begin_instances_rec
iter.targets = "A"
while !iter.at_end?
  puts "Instance of #{iter.inst_cell.name} in #{cell.name}: " + (iter.dtrans * iter.inst_dtrans).to_s
  iter.next
end

# or shorter:
cell.begin_instances_rec.each do |iter|
  puts "Instance of #{iter.inst_cell.name} in #{cell.name}: " + (iter.dtrans * iter.inst_dtrans).to_s
end
```

Here, a target cell is specified which confines the search to instances of this particular cell. 'iter.dtrans' gives us the accumulated transformation of all parents up to the top cell. 'iter.inst_dtrans' gives us the transformation from the current instance. 'iter.inst_cell' finally gives us the target cell of the current instance (which is always 'A' in our case).

Cell offers three methods to get these iterators: begin_instances_rec, begin_instances_rec_touching and begin_instances_rec_overlapping. Cell#begin_instances_rec will deliver a standard recursive instance iterator which starts from the given cell and iterates over all child cells. Cell#begin_instances_rec_touching creates a RecursiveInstanceIterator which delivers the instances whose bounding boxed touch the given search box. Cell#begin_instances_rec_overlapping gives an iterator which delivers all instances whose bounding box overlaps the search box.

A RecursiveInstanceIterator object can also be created directly, like this:

```
iter = RBA::RecursiveInstanceIterator::new(layout, cell [, options ])
```

"layout" is the layout object, "cell" the Cell object of the initial cell.

The recursive instance iterator can be confined to a maximum hierarchy depth. By using max_depth=, the iterator will restrict the search depth to the given depth in the cell tree. In the same way, the iterator can be configured to start from a certain hierarchy depth using min_depth=. The hierarchy depth always applies to the parent of the instances iterated.

In addition, the recursive instance iterator supports selection and exclusion of subtrees. For that purpose it keeps flags per cell telling it for which cells to turn instance delivery on and off. The select_cells method sets the "start delivery" flag while unselect_cells sets the "stop delivery" flag. In effect, using unselect_cells will exclude that cell plus the subtree from delivery. Parts of that subtree can be turned on again using select_cells. For the cells selected that way, the instances of these cells and their child cells are delivered, even if their parent was unselected.

To get instances from a specific cell, i.e. "MACRO" plus its child cells, unselect the top cell first and the select the desired cell again:

```
# deliver all instances inside "MACRO" and the sub-hierarchy:
iter = RBA::RecursiveInstanceIterator::new(layout, cell)
iter.unselect_cells(cell.cell_index)
iter.select_cells("MACRO")
```

```
...
```

The unselect_all_cells and select_all_cells methods turn on the "stop" and "start" flag for all cells respectively. If you use unselect_all_cells and use select_cells for a specific cell, the iterator will deliver only the instances of the selected cell, not its children. Those are still unselected by unselect_all_cells:

```
# deliver all instance inside "MACRO" but not of child cells:
iter = RBA::RecursiveInstanceIterator::new(layout, cell)
iter.unselect_all_cells
iter.select_cells("MACRO")
...
```

Cell selection is done using cell indexes or glob pattern. Glob pattern are equivalent to the usual file name wildcards used on various command line shells. For example "A*" matches all cells starting with an "A". The curly brace notation and character classes are supported as well. For example "C{125,512}" matches "C125" and "C512" and "[ABC]*" matches all cells starting with an "A", a "B" or "C". "[^ABC]*" matches all cells not starting with one of that letters.

To confine instance iteration to instances of certain cells, use the targets feature:

```
# deliver all instance of "INV1":
iter = RBA::RecursiveInstanceIterator::new(layout, cell)
iter.targets = "INV1"
...
```

Targets can be specified either as lists of cell indexes or through a glob pattern.

Instances are always delivered depth-first with child instances before their parents. A default recursive instance iterator will first deliver leaf cells, followed by the parent of these cells.

When a search region is used, instances whose bounding box touch or overlap (depending on 'overlapping' flag) will be reported. The instance bounding box taken as reference is computed using all layers of the layout.

The iterator will deliver the individual elements of instance arrays, confined to the search region if one is given. Consequently the return value (current_inst_element) is an InstElement object which is basically a combination of an Instance object and information about the current array element. inst_cell, inst_trans and inst_dtrans are methods provided for convenience to access the current array member's transformation and the target cell of the current instance.

The RecursiveInstanceIterator class has been introduced in version 0.27.

Public constructors

new RecursiveInstanceIterator ptr	**new**	(const Layout layout, const Cell cell)	Creates a recursive instance iterator.
new RecursiveInstanceIterator ptr	**new**	(const Layout layout, const Cell cell, const Box box, bool overlapping = false)	Creates a recursive instance iterator with a search region.
new RecursiveInstanceIterator ptr	**new**	(const Layout layout, const Cell cell, const Region region, bool overlapping)	Creates a recursive instance iterator with a search region.

Public methods

[const]	bool	**!=**	(const RecursiveInstar other)	Comparison of iterators - inequality
[const]	bool	**==**	(const RecursiveInstanceIterator other)	Comparison of iterators - equality
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	bool	**all_targets_enabled?**		Gets a value indicating whether instances of all cells are reported
	void	**assign**	(const RecursiveInstanceIterator other)	Assigns another object to self
[const]	bool	**at_end?**		End of iterator predicate
[const]	const Cell ptr	**cell**		Gets the cell the current instance sits in
[const]	unsigned int	**cell_index**		Gets the index of the cell the current instance sits in
[const]	Region	**complex_region**		Gets the complex region that this iterator is using
	void	**confine_region**	(const Box box_region)	Confines the region that this iterator is iterating over
	void	**confine_region**	(const Region complex_region)	Confines the region that this iterator is iterating over
[const]	InstElement	**current_inst_element**		Gets the current instance
[const]	DCplxTrans	**dtrans**		Gets the accumulated transformation of the current instance parent cell to the top cell
[const]	new RecursiveInstanceItera ptr	**dup**		Creates a copy of self

[iter]	RecursiveInstanceIterator	**each**		Native iteration
	void	**enable_all_targets**		Enables 'all targets' mode in which instances of all cells are reported
[const]	Cell ptr	**inst_cell**		Gets the target cell of the current instance
[const]	DCplxTrans	**inst_dtrans**		Gets the micron-unit transformation of the current instance
[const]	ICplxTrans	**inst_trans**		Gets the integer-unit transformation of the current instance
[const]	const Layout ptr	**layout**		Gets the layout this iterator is connected to
[const]	int	**max_depth**		Gets the maximum hierarchy depth
	void	**max_depth=**	(int depth)	Specifies the maximum hierarchy depth to look into
[const]	int	**min_depth**		Gets the minimum hierarchy depth
	void	**min_depth=**	(int depth)	Specifies the minimum hierarchy depth to look into
	void	**next**		Increments the iterator
	void	**overlapping=**	(bool region)	Sets a flag indicating whether overlapping instances are selected when a region is used
[const]	bool	**overlapping?**		Gets a flag indicating whether overlapping instances are selected when a region is used
[const]	InstElement[]	**path**		Gets the instantiation path of the instance addressed currently
[const]	Box	**region**		Gets the basic region that this iterator is using
	void	**region=**	(const Box box_region)	Sets the rectangular region that this iterator is iterating over
	void	**region=**	(const Region complex_region)	Sets the complex region that this iterator is using
	void	**reset**		Resets the iterator to the initial state
	void	**reset_selection**		Resets the selection to the default state
	void	**select_all_cells**		Selects all cells.
	void	**select_cells**	(unsigned int[] cells)	Unselects the given cells.
	void	**select_cells**	(string cells)	Unselects the given cells.
[const]	unsigned int[]	**targets**		Gets the list of target cells

	void	**targets=**	(unsigned int[] cells)	Specifies the target cells.
	void	**targets=**	(string cells)	Specifies the target cells.
[const]	const Cell ptr	**top_cell**		Gets the top cell this iterator is connected to
[const]	ICplxTrans	**trans**		Gets the accumulated transformation of the current instance parent cell to the top cell
	void	**unselect_all_cells**		Unselects all cells.
	void	**unselect_cells**	(unsigned int[] cells)	Unselects the given cells.
	void	**unselect_cells**	(string cells)	Unselects the given cells.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const RecursiveInstanceIterator other)

Description: Comparison of iterators - inequality

Two iterators are not equal if they do not point to the same instance.

==

Signature: *[const]* bool **==** (const RecursiveInstanceIterator other)

Description: Comparison of iterators - equality

Two iterators are equal if they point to the same instance.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

all_targets_enabled?

Signature: *[const]* bool **all_targets_enabled?**

Description: Gets a value indicating whether instances of all cells are reported

See targets= for a description of the target cell concept.

assign

Signature: void **assign** (const RecursiveInstanceIterator other)

Description: Assigns another object to self

at_end?

Signature: *[const]* bool **at_end?**

Description: End of iterator predicate

Returns true, if the iterator is at the end of the sequence

cell

Signature: *[const]* const Cell ptr **cell**

Description: Gets the cell the current instance sits in

cell_index

Signature: *[const]* unsigned int **cell_index**

Description: Gets the index of the cell the current instance sits in

This is equivalent to 'cell.cell_index'.

complex_region

Signature: *[const]* Region **complex_region**

Description: Gets the complex region that this iterator is using

The complex region is the effective region (a Region object) that the iterator is selecting from the layout. This region can be a single box or a complex region.

confine_region

(1) Signature: void **confine_region** (const Box box_region)

Description: Confines the region that this iterator is iterating over

This method is similar to setting the region (see region=), but will confine any region (complex or simple) already set. Essentially it does a logical AND operation between the existing and given region. Hence this method can only reduce a region, not extend it.

(2) Signature: void **confine_region** (const Region complex_region)

Description: Confines the region that this iterator is iterating over

This method is similar to setting the region (see region=), but will confine any region (complex or simple) already set. Essentially it does a logical AND operation between the existing and given region. Hence this method can only reduce a region, not extend it.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

current_inst_element

Signature: *[const]* InstElement **current_inst_element**

Description: Gets the current instance

This is the instance/array element the iterator currently refers to. This is a InstElement object representing the current instance and the array element the iterator currently points at.

See inst_trans, inst_dtrans and inst_cell for convenience methods to access the details of the current element.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dtrans

Signature: *[const]* DCplxTrans **dtrans**

Description: Gets the accumulated transformation of the current instance parent cell to the top cell

This transformation represents how the current instance is seen in the top cell. This version returns the micron-unit transformation.

dup

Signature: *[const]* new RecursiveInstanceIterator ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each

Signature: *[iter]* RecursiveInstanceIterator **each**

Description: Native iteration

This method enables native iteration, e.g.

```
iter = ... # RecursiveInstanceIterator
iter.each do |i|
   ... i is the iterator itself
end
```

This is slightly more convenient than the 'at_end' .. 'next' loop.

This feature has been introduced in version 0.28.

Python specific notes:
This method enables iteration of the object.

enable_all_targets

Signature: void **enable_all_targets**

Description: Enables 'all targets' mode in which instances of all cells are reported

See targets= for a description of the target cell concept.

inst_cell

Signature: *[const]* Cell ptr **inst_cell**

Description: Gets the target cell of the current instance

This is the cell the current instance refers to. It is one of the targets if a target list is given.

inst_dtrans

Signature: *[const]* DCplxTrans **inst_dtrans**

Description: Gets the micron-unit transformation of the current instance

This is the transformation of the current instance inside its parent. 'dtrans * inst_dtrans' gives the full micron-unit transformation how the current cell is seen in the top cell. See also inst_trans and inst_cell.

inst_trans

Signature: *[const]* ICplxTrans **inst_trans**

Description: Gets the integer-unit transformation of the current instance

This is the transformation of the current instance inside its parent. 'trans * inst_trans' gives the full transformation how the current cell is seen in the top cell. See also inst_dtrans and inst_cell.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

layout

Signature: *[const]* const Layout ptr **layout**

Description: Gets the layout this iterator is connected to

max_depth

Signature: *[const]* int **max_depth**

Description: Gets the maximum hierarchy depth

See max_depth= for a description of that attribute.

Python specific notes:
The object exposes a readable attribute 'max_depth'. This is the getter.

max_depth=

Signature: void **max_depth=** (int depth)

Description: Specifies the maximum hierarchy depth to look into

A depth of 0 instructs the iterator to deliver only instances from the initial cell. A higher depth instructs the iterator to look deeper. The depth must be specified before the instances are being retrieved.

Python specific notes:
The object exposes a writable attribute 'max_depth'. This is the setter.

min_depth

Signature: *[const]* int **min_depth**

Description: Gets the minimum hierarchy depth

See min_depth= for a description of that attribute.

Python specific notes:
The object exposes a readable attribute 'min_depth'. This is the getter.

min_depth=

Signature: void **min_depth=** (int depth)

Description: Specifies the minimum hierarchy depth to look into

A depth of 0 instructs the iterator to deliver instances from the top level. 1 instructs to deliver instances from the first child level. The minimum depth must be specified before the instances are being retrieved.

Python specific notes:
The object exposes a writable attribute 'min_depth'. This is the setter.

new

(1) Signature: *[static]* new RecursiveInstanceIterator ptr **new** (const Layout layout, const Cell cell)

Description: Creates a recursive instance iterator.

layout:	The layout which shall be iterated
cell:	The initial cell which shall be iterated (including its children)
layer:	The layer (index) from which the shapes are taken

This constructor creates a new recursive instance iterator which delivers the instances of the given cell plus its children.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new RecursiveInstanceIterator ptr **new** (const Layout layout, const Cell cell, const Box box, bool overlapping = false)

Description: Creates a recursive instance iterator with a search region.

layout:	The layout which shall be iterated

cell:	The initial cell which shall be iterated (including its children)
box:	The search region
overlapping:	If set to true, instances overlapping the search region are reported, otherwise touching is sufficient

This constructor creates a new recursive instance iterator which delivers the instances of the given cell plus its children.

The search is confined to the region given by the "box" parameter. If "overlapping" is true, instances whose bounding box is overlapping the search region are reported. If "overlapping" is false, instances whose bounding box touches the search region are reported. The bounding box of instances is measured taking all layers of the target cell into account.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new RecursiveInstanceIterator ptr **new** (const Layout layout, const Cell cell, const Region region, bool overlapping)

Description: Creates a recursive instance iterator with a search region.

layout:	The layout which shall be iterated
cell:	The initial cell which shall be iterated (including its children)
region:	The search region
overlapping:	If set to true, instances overlapping the search region are reported, otherwise touching is sufficient

This constructor creates a new recursive instance iterator which delivers the instances of the given cell plus its children.

The search is confined to the region given by the "region" parameter. The region needs to be a rectilinear region. If "overlapping" is true, instances whose bounding box is overlapping the search region are reported. If "overlapping" is false, instances whose bounding box touches the search region are reported. The bounding box of instances is measured taking all layers of the target cell into account.

Python specific notes:
This method is the default initializer of the object.

next

Signature: void **next**

Description: Increments the iterator

This moves the iterator to the next instance inside the search scope.

overlapping=

Signature: void **overlapping=** (bool region)

Description: Sets a flag indicating whether overlapping instances are selected when a region is used

If this flag is false, instances touching the search region are returned.

Python specific notes:
The object exposes a writable attribute 'overlapping'. This is the setter.

overlapping?

Signature: *[const]* bool **overlapping?**

Description: Gets a flag indicating whether overlapping instances are selected when a region is used

Python specific notes:
The object exposes a readable attribute 'overlapping'. This is the getter.

path

Signature: *[const]* InstElement[] **path**

Description: Gets the instantiation path of the instance addressed currently

This attribute is a sequence of InstElement objects describing the cell instance path from the initial cell to the current instance. The path is empty if the current instance is in the top cell.

region

Signature: *[const]* Box **region**

Description: Gets the basic region that this iterator is using

The basic region is the overall box the region iterator iterates over. There may be an additional complex region that confines the region iterator. See complex_region for this attribute.

Python specific notes:
The object exposes a readable attribute 'region'. This is the getter.

region=

(1) Signature: void **region=** (const Box box_region)

Description: Sets the rectangular region that this iterator is iterating over

See region for a description of this attribute. Setting a simple region will reset the complex region to a rectangle and reset the iterator to the beginning of the sequence.

Python specific notes:
The object exposes a writable attribute 'region'. This is the setter.

(2) Signature: void **region=** (const Region complex_region)

Description: Sets the complex region that this iterator is using

See complex_region for a description of this attribute. Setting the complex region will reset the basic region (see region) to the bounding box of the complex region and reset the iterator to the beginning of the sequence.

Python specific notes:
The object exposes a writable attribute 'region'. This is the setter.

reset

Signature: void **reset**

Description: Resets the iterator to the initial state

reset_selection

Signature: void **reset_selection**

Description: Resets the selection to the default state

In the initial state, the top cell and its children are selected. Child cells can be switched on and off together with their sub-hierarchy using select_cells and unselect_cells.

This method will also reset the iterator.

select_all_cells

Signature: void **select_all_cells**

Description: Selects all cells.

This method will set the "selected" mark on all cells. The effect is that subsequent calls of unselect_cells will unselect only the specified cells, not their children, because they are still unselected.

This method will also reset the iterator.

select_cells

(1) Signature: void **select_cells** (unsigned int[] cells)

Description: Unselects the given cells.

This method will sets the "selected" mark on the given cells. That means that these cells or their child cells are visited, unless they are marked as "unselected" again with the unselect_cells method.

The cells are given as a list of cell indexes.

This method will also reset the iterator.

(2) Signature: void **select_cells** (string cells)

Description: Unselects the given cells.

This method will sets the "selected" mark on the given cells. That means that these cells or their child cells are visited, unless they are marked as "unselected" again with the unselect_cells method.

The cells are given as a glob pattern. A glob pattern follows the syntax of file names on the shell (i.e. "A*" are all cells starting with a letter "A").

This method will also reset the iterator.

targets

Signature: *[const]* unsigned int[] **targets**

Description: Gets the list of target cells

See targets= for a description of the target cell concept. This method returns a list of cell indexes of the selected target cells.

Python specific notes:
The object exposes a readable attribute 'targets'. This is the getter.

targets=

(1) Signature: void **targets=** (unsigned int[] cells)

Description: Specifies the target cells.

If target cells are specified, only instances of these cells are delivered. This version takes a list of cell indexes for the targets. By default, no target cell list is present and the instances of all cells are delivered by the iterator. See all_targets_enabled? and enable_all_targets for a description of this mode. Once a target list is specified, the iteration is confined to the cells from this list. The cells are given as a list of cell indexes.

This method will also reset the iterator.

Python specific notes:
The object exposes a writable attribute 'targets'. This is the setter.

(2) Signature: void **targets=** (string cells)

Description: Specifies the target cells.

If target cells are specified, only instances of these cells are delivered. This version takes a cell list as a glob pattern. A glob pattern follows the syntax of file names on the shell (i.e. "A*" are all cells starting with a letter "A"). Use the curly-bracket notation to list different cells, e.g "{A,B,C}" for cells A, B and C.

By default, no target cell list is present and the instances of all cells are delivered by the iterator. See all_targets_enabled? and enable_all_targets for a description of this mode. Once a target list is specified, the iteration is confined to the cells from this list. The cells are given as a list of cell indexes.

This method will also reset the iterator.

Python specific notes:
The object exposes a writable attribute 'targets'. This is the setter.

top_cell

Signature: *[const]* const Cell ptr **top_cell**

Description: Gets the top cell this iterator is connected to

trans

Signature: *[const]* ICplxTrans **trans**

Description: Gets the accumulated transformation of the current instance parent cell to the top cell

This transformation represents how the current instance is seen in the top cell.

unselect_all_cells

Signature: void **unselect_all_cells**

Description: Unselects all cells.

This method will set the "unselected" mark on all cells. The effect is that subsequent calls of select_cells will select only the specified cells, not their children, because they are still unselected.

This method will also reset the iterator.

unselect_cells

(1) Signature: void **unselect_cells** (unsigned int[] cells)

Description: Unselects the given cells.

This method will sets the "unselected" mark on the given cells. That means that these cells or their child cells will not be visited, unless they are marked as "selected" again with the select_cells method.

The cells are given as a list of cell indexes.

This method will also reset the iterator.

(2) Signature: void **unselect_cells** (string cells)

Description: Unselects the given cells.

This method will sets the "unselected" mark on the given cells. That means that these cells or their child cells will not be visited, unless they are marked as "selected" again with the select_cells method.

The cells are given as a glob pattern. A glob pattern follows the syntax of file names on the shell (i.e. "A*" are all cells starting with a letter "A").

This method will also reset the iterator.

4.82. API reference - Class RecursiveShapeIterator

Notation used in Ruby API documentation

Module: db

Description: An iterator delivering shapes recursively

The iterator can be obtained from a cell, a layer and optionally a region. It simplifies retrieval of shapes from a geometrical region while considering subcells as well. Some options can be specified in addition, i.e. the level to which to look into or shape classes and shape properties. The shapes are retrieved by using the shape method, next moves to the next shape and at_end tells, if the iterator has move shapes to deliver.

This is some sample code:

```
# print the polygon-like objects as seen from the initial cell "cell"
iter = cell.begin_shapes_rec(layer)
while !iter.at_end?
  if iter.shape.renders_polygon?
    polygon = iter.shape.polygon.transformed(iter.itrans)
    puts "In cell #{iter.cell.name}: " + polygon.to_s
  end
  iter.next
end

# or shorter:
cell.begin_shapes_rec(layer).each do |iter|
  if iter.shape.renders_polygon?
    polygon = iter.shape.polygon.transformed(iter.itrans)
    puts "In cell #{iter.cell.name}: " + polygon.to_s
  end
end
```

Cell offers three methods to get these iterators: begin_shapes_rec, begin_shapes_rec_touching and begin_shapes_rec_overlapping. Cell#begin_shapes_rec will deliver a standard recursive shape iterator which starts from the given cell and iterates over all child cells. Cell#begin_shapes_rec_touching delivers a RecursiveShapeIterator which delivers the shapes whose bounding boxed touch the given search box. Cell#begin_shapes_rec_overlapping delivers all shapes whose bounding box overlaps the search box.

A RecursiveShapeIterator object can also be created explicitly. This allows some more options, i.e. using multiple layers. A multi-layer recursive shape iterator can be created like this:

```
iter = RBA::RecursiveShapeIterator::new(layout, cell, [ layer_index1, layer_index2 .. ])
```

"layout" is the layout object, "cell" the RBA::Cell object of the initial cell. layer_index1 etc. are the layer indexes of the layers to get the shapes from. While iterating, RecursiveShapeIterator#layer delivers the layer index of the current shape.

The recursive shape iterator can be confined to a maximum hierarchy depth. By using max_depth=, the iterator will restrict the search depth to the given depth in the cell tree.

In addition, the recursive shape iterator supports selection and exclusion of subtrees. For that purpose it keeps flags per cell telling it for which cells to turn shape delivery on and off. The select_cells method sets the "start delivery" flag while unselect_cells sets the "stop delivery" flag. In effect, using unselect_cells will exclude that cell plus the subtree from delivery. Parts of that subtree can be turned on again using select_cells. For the cells selected that way, the shapes of these cells and their child cells are delivered, even if their parent was unselected.

To get shapes from a specific cell, i.e. "MACRO" plus its child cells, unselect the top cell first and the select the desired cell again:

```
# deliver all shapes inside "MACRO" and the sub-hierarchy:
iter = RBA::RecursiveShapeIterator::new(layout, cell, layer)
iter.unselect_cells(cell.cell_index)
```

```
iter.select_cells("MACRO")
```

Note that if "MACRO" uses library cells for example which are used otherwise as well, the iterator will only deliver the shapes for those instances belonging to "MACRO" (directly or indirectly), not those for other instances of these library cells.

The unselect_all_cells and select_all_cells methods turn on the "stop" and "start" flag for all cells respectively. If you use unselect_all_cells and use select_cells for a specific cell, the iterator will deliver only the shapes of the selected cell, not its children. Those are still unselected by unselect_all_cells:

```
# deliver all shapes of "MACRO" but not of child cells:
iter = RBA::RecursiveShapeIterator::new(layout, cell, layer)
iter.unselect_all_cells
iter.select_cells("MACRO")
```

Cell selection is done using cell indexes or glob pattern. Glob pattern are equivalent to the usual file name wildcards used on various command line shells. For example "A*" matches all cells starting with an "A". The curly brace notation and character classes are supported as well. For example "C{125,512}" matches "C125" and "C512" and "[ABC]*" matches all cells starting with an "A", a "B" or "C". "[^ABC]*" matches all cells not starting with one of that letters.

The RecursiveShapeIterator class has been introduced in version 0.18 and has been extended substantially in 0.23.

Public constructors

new RecursiveShapeIterator ptr	**new**	(const Layout layout, const Cell cell, unsigned int layer)	Creates a recursive, single-layer shape iterator.
new RecursiveShapeIterator ptr	**new**	(const Layout layout, const Cell cell, unsigned int[] layers)	Creates a recursive, multi-layer shape iterator.
new RecursiveShapeIterator ptr	**new**	(const Layout layout, const Cell cell, unsigned int layer, const Box box, bool overlapping = false)	Creates a recursive, single-layer shape iterator with a region.
new RecursiveShapeIterator ptr	**new**	(const Layout layout, const Cell cell, unsigned int layer, const Region region, bool overlapping = false)	Creates a recursive, single-layer shape iterator with a region.
new RecursiveShapeIterator ptr	**new**	(const Layout layout, const Cell cell, unsigned int[] layers, const Box box, bool overlapping = false)	Creates a recursive, multi-layer shape iterator with a region.
new RecursiveShapeIterator ptr	**new**	(const Layout layout, const Cell cell, unsigned int[] layers, const Region region, bool overlapping = false)	Creates a recursive, multi-layer shape iterator with a region.

Public methods

[const]	bool	**!=**	(const RecursiveShape other)	Comparison of iterators - inequality
[const]	bool	**==**	(const RecursiveShapeIterator other)	Comparison of iterators - equality
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	DCplxTrans	**always_apply_dtrans**		Gets the global transformation if at top level, unity otherwise (micrometer-unit version)
[const]	ICplxTrans	**always_apply_trans**		Gets the global transformation if at top level, unity otherwise
	void	**assign**	(const RecursiveShape other)	Assigns another object to self
[const]	bool	**at_end?**		End of iterator predicate
[const]	const Cell ptr	**cell**		Gets the current cell's object
[const]	unsigned int	**cell_index**		Gets the current cell's index
[const]	Region	**complex_region**		Gets the complex region that this iterator is using
	void	**confine_region**	(const Box box_region)	Confines the region that this iterator is iterating over
	void	**confine_region**	(const Region complex_region)	Confines the region that this iterator is iterating over
[const]	DCplxTrans	**dtrans**		Gets the transformation into the initial cell applicable for floating point types
[const]	new RecursiveShapeIterator ptr	**dup**		Creates a copy of self
[iter]	RecursiveShapeIterator	**each**		Native iteration

	void	**enable_properties**		Enables properties for the given iterator.
	void	**filter_properties**	(variant[] keys)	Filters properties by certain keys.
	void	**for_merged_input=**	(bool flag)	Sets a flag indicating whether iterator optimizes for merged input
[const]	bool	**for_merged_input?**		Gets a flag indicating whether iterator optimizes for merged input
[const]	DCplxTrans	**global_dtrans**		Gets the global transformation to apply to all shapes delivered (in micrometer units)
	void	**global_dtrans=**	(const DCplxTrans t)	Sets the global transformation to apply to all shapes delivered (transformation in micrometer units)
[const]	ICplxTrans	**global_trans**		Gets the global transformation to apply to all shapes delivered
	void	**global_trans=**	(const ICplxTrans t)	Sets the global transformation to apply to all shapes delivered
[const]	unsigned int	**layer**		Returns the layer index where the current shape is coming from.
[const]	const Layout ptr	**layout**		Gets the layout this iterator is connected to
	void	**map_properties**	(map<variant,va key_map)	Maps properties by name key.
[const]	int	**max_depth**		Gets the maximum hierarchy depth
	void	**max_depth=**	(int depth)	Specifies the maximum hierarchy depth to look into
[const]	int	**min_depth**		Gets the minimum hierarchy depth
	void	**min_depth=**	(int depth)	Specifies the minimum hierarchy depth to look into
	void	**next**		Increments the iterator
	void	**overlapping=**	(bool flag)	Sets a flag indicating whether overlapping shapes are selected when a region is used
[const]	bool	**overlapping?**		Gets a flag indicating whether overlapping shapes are selected when a region is used
[const]	InstElement[]	**path**		Gets the instantiation path of the shape addressed currently
[const]	unsigned long	**prop_id**		Gets the effective properties ID
[const]	Box	**region**		Gets the basic region that this iterator is using

	void	**region=**	(const Box box_region)	Sets the rectangular region that this iterator is iterating over
	void	**region=**	(const Region complex_region)	Sets the complex region that this iterator is using
	void	**remove_properties**		Removes properties for the given container.
	void	**reset**		Resets the iterator to the initial state
	void	**reset_selection**		Resets the selection to the default state
	void	**select_all_cells**		Selects all cells.
	void	**select_cells**	(unsigned int[] cells)	Unselects the given cells.
	void	**select_cells**	(string cells)	Unselects the given cells.
[const]	Shape	**shape**		Gets the current shape
[const]	unsigned int	**shape_flags**		Gets the shape selection flags
	void	**shape_flags=**	(unsigned int flags)	Specifies the shape selection flags
[const]	const Cell ptr	**top_cell**		Gets the top cell this iterator is connected to
[const]	ICplxTrans	**trans**		Gets the current transformation by which the shapes must be transformed into the initial cell
	void	**unselect_all_cells**		Unselects all cells.
	void	**unselect_cells**	(unsigned int[] cells)	Unselects the given cells.
	void	**unselect_cells**	(string cells)	Unselects the given cells.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead
[const]	ICplxTrans	**itrans**	Use of this method is deprecated. Use trans instead

Detailed description

!=

Signature: *[const]* bool **!=** (const RecursiveShapeIterator other)
Description: Comparison of iterators - inequality

	Two iterators are not equal if they do not point to the same shape.
==	**Signature**: *[const]* bool **==** (const RecursiveShapeIterator other) **Description**: Comparison of iterators - equality Two iterators are equal if they point to the same shape.
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.

always_apply_dtrans

Signature: *[const]* DCplxTrans **always_apply_dtrans**

Description: Gets the global transformation if at top level, unity otherwise (micrometer-unit version)

As the global transformation is only applicable on top level, use this method to transform shapes and instances into their local (cell-level) version while considering the global transformation properly.

This method has been introduced in version 0.27.

always_apply_trans

Signature: *[const]* ICplxTrans **always_apply_trans**

Description: Gets the global transformation if at top level, unity otherwise

As the global transformation is only applicable on top level, use this method to transform shapes and instances into their local (cell-level) version while considering the global transformation properly.

This method has been introduced in version 0.27.

assign

Signature: void **assign** (const RecursiveShapeIterator other)

Description: Assigns another object to self

at_end?

Signature: *[const]* bool **at_end?**

Description: End of iterator predicate

Returns true, if the iterator is at the end of the sequence

cell

Signature: *[const]* const Cell ptr **cell**

Description: Gets the current cell's object

This method has been introduced in version 0.23.

cell_index

Signature: *[const]* unsigned int **cell_index**

Description: Gets the current cell's index

complex_region

Signature: *[const]* Region **complex_region**

Description: Gets the complex region that this iterator is using

The complex region is the effective region (a Region object) that the iterator is selecting from the layout layers. This region can be a single box or a complex region.

This method has been introduced in version 0.25.

confine_region

(1) Signature: void **confine_region** (const Box box_region)

Description: Confines the region that this iterator is iterating over

This method is similar to setting the region (see region=), but will confine any region (complex or simple) already set. Essentially it does a logical AND operation between the existing and given region. Hence this method can only reduce a region, not extend it.

This method has been introduced in version 0.25.

(2) Signature: void **confine_region** (const Region complex_region)

Description: Confines the region that this iterator is iterating over

This method is similar to setting the region (see region=), but will confine any region (complex or simple) already set. Essentially it does a logical AND operation between the existing and given region. Hence this method can only reduce a region, not extend it.

This method has been introduced in version 0.25.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dtrans

Signature: *[const]* DCplxTrans **dtrans**

Description: Gets the transformation into the initial cell applicable for floating point types

This transformation corresponds to the one delivered by trans, but is applicable for the floating-point shape types in micron unit space.

This method has been introduced in version 0.25.3.

dup

Signature: *[const]* new RecursiveShapeIterator ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each

Signature: *[iter]* RecursiveShapeIterator **each**

Description: Native iteration

This method enables native iteration, e.g.

```
iter = ... # RecursiveShapeIterator
iter.each do |i|
   ... i is the iterator itself
end
```

This is slightly more convenient than the 'at_end' .. 'next' loop.

This feature has been introduced in version 0.28.

Python specific notes:
This method enables iteration of the object.

enable_properties

Signature: void **enable_properties**

Description: Enables properties for the given iterator.

Afer enabling properties, prop_id will deliver the effective properties ID for the current shape.
By default, properties are not enabled and prop_id will always return 0 (no properties attached).
Alternatively you can apply filter_properties or map_properties to enable properties with a specific name key.

Note that property filters/mappers are additive and act in addition (after) the currently installed filter.

This feature has been introduced in version 0.28.4.

filter_properties

Signature: void **filter_properties** (variant[] keys)

Description: Filters properties by certain keys.

Calling this method will reduce the properties to values with name keys from the 'keys' list. As a side effect, this method enables properties. As with enable_properties or remove_properties, this filter has an effect on the value returned by prop_id, not on the properties ID attached to the shape directly.

Note that property filters/mappers are additive and act in addition (after) the currently installed filter.

This feature has been introduced in version 0.28.4.

for_merged_input=

Signature: void **for_merged_input=** (bool flag)

Description: Sets a flag indicating whether iterator optimizes for merged input

If this flag is set to true, the iterator is allowed to skip shapes it deems irrelevant because they are covered entirely by other shapes. This allows shortcutting hierarchy traversal in some cases.

This method has been introduced in version 0.29.

Python specific notes:
The object exposes a writable attribute 'for_merged_input'. This is the setter.

for_merged_input?

Signature: *[const]* bool **for_merged_input?**

Description: Gets a flag indicating whether iterator optimizes for merged input

see for_merged_input= for details of this attribute.

This method has been introduced in version 0.29.

Python specific notes:
The object exposes a readable attribute 'for_merged_input'. This is the getter.

global_dtrans

Signature: *[const]* DCplxTrans **global_dtrans**

Description: Gets the global transformation to apply to all shapes delivered (in micrometer units)

See also global_dtrans=.

This method has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'global_dtrans'. This is the getter.

global_dtrans=

Signature: void **global_dtrans=** (const DCplxTrans t)

Description: Sets the global transformation to apply to all shapes delivered (transformation in micrometer units)

The global transformation will be applied to all shapes delivered by biasing the "trans" attribute. The search regions apply to the coordinate space after global transformation.

This method has been introduced in version 0.27.

Python specific notes:

The object exposes a writable attribute 'global_dtrans'. This is the setter.

global_trans

Signature: *[const]* ICplxTrans **global_trans**

Description: Gets the global transformation to apply to all shapes delivered

See also global_trans=.

This method has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'global_trans'. This is the getter.

global_trans=

Signature: void **global_trans=** (const ICplxTrans t)

Description: Sets the global transformation to apply to all shapes delivered

The global transformation will be applied to all shapes delivered by biasing the "trans" attribute. The search regions apply to the coordinate space after global transformation.

This method has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'global_trans'. This is the setter.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

itrans

Signature: *[const]* ICplxTrans **itrans**

Description: Gets the current transformation by which the shapes must be transformed into the initial cell

Use of this method is deprecated. Use trans instead

The shapes delivered are not transformed. Instead, this transformation must be applied to get the shape in the coordinate system of the top cell.

Starting with version 0.25, this transformation is a int-to-int transformation the 'itrans' method which was providing this transformation before is deprecated.

layer

Signature: *[const]* unsigned int **layer**

Description: Returns the layer index where the current shape is coming from.

This method has been introduced in version 0.23.

layout

Signature: *[const]* const Layout ptr **layout**

Description: Gets the layout this iterator is connected to

This method has been introduced in version 0.23.

map_properties

Signature: void **map_properties** (map<variant,variant> key_map)

Description: Maps properties by name key.

Calling this method will reduce the properties to values with name keys from the 'keys' hash and renames the properties. Property values with keys not listed in the key map will be removed. As a side effect, this method enables properties. As with enable_properties or remove_properties, this

filter has an effect on the value returned by prop_id, not on the properties ID attached to the shape directly.

Note that property filters/mappers are additive and act in addition (after) the currently installed filter.

This feature has been introduced in version 0.28.4.

max_depth

Signature: *[const]* int **max_depth**

Description: Gets the maximum hierarchy depth

See max_depth= for a description of that attribute.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'max_depth'. This is the getter.

max_depth=

Signature: void **max_depth=** (int depth)

Description: Specifies the maximum hierarchy depth to look into

A depth of 0 instructs the iterator to deliver only shapes from the initial cell. The depth must be specified before the shapes are being retrieved. Setting the depth resets the iterator.

Python specific notes:
The object exposes a writable attribute 'max_depth'. This is the setter.

min_depth

Signature: *[const]* int **min_depth**

Description: Gets the minimum hierarchy depth

See min_depth= for a description of that attribute.

This method has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'min_depth'. This is the getter.

min_depth=

Signature: void **min_depth=** (int depth)

Description: Specifies the minimum hierarchy depth to look into

A depth of 0 instructs the iterator to deliver shapes from the top level. 1 instructs to deliver shapes from the first child level. The minimum depth must be specified before the shapes are being retrieved.

This method has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'min_depth'. This is the setter.

new

(1) Signature: *[static]* new RecursiveShapeIterator ptr **new** (const Layout layout, const Cell cell, unsigned int layer)

Description: Creates a recursive, single-layer shape iterator.

layout:	The layout which shall be iterated
cell:	The initial cell which shall be iterated (including its children)
layer:	The layer (index) from which the shapes are taken

This constructor creates a new recursive shape iterator which delivers the shapes of the given cell plus its children from the layer given by the layer index in the "layer" parameter.

This constructor has been introduced in version 0.23.

Python specific notes:

This method is the default initializer of the object.

(2) Signature: *[static]* new RecursiveShapeIterator ptr **new** (const Layout layout, const Cell cell, unsigned int[] layers)

Description: Creates a recursive, multi-layer shape iterator.

layout:	The layout which shall be iterated
cell:	The initial cell which shall be iterated (including its children)
layers:	The layer indexes from which the shapes are taken

This constructor creates a new recursive shape iterator which delivers the shapes of the given cell plus its children from the layers given by the layer indexes in the "layers" parameter. While iterating use the layer method to retrieve the layer of the current shape.

This constructor has been introduced in version 0.23.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new RecursiveShapeIterator ptr **new** (const Layout layout, const Cell cell, unsigned int layer, const Box box, bool overlapping = false)

Description: Creates a recursive, single-layer shape iterator with a region.

layout:	The layout which shall be iterated
cell:	The initial cell which shall be iterated (including its children)
layer:	The layer (index) from which the shapes are taken
box:	The search region
overlapping:	If set to true, shapes overlapping the search region are reported, otherwise touching is sufficient

This constructor creates a new recursive shape iterator which delivers the shapes of the given cell plus its children from the layer given by the layer index in the "layer" parameter.

The search is confined to the region given by the "box" parameter. If "overlapping" is true, shapes whose bounding box is overlapping the search region are reported. If "overlapping" is false, shapes whose bounding box touches the search region are reported.

This constructor has been introduced in version 0.23. The 'overlapping' parameter has been made optional in version 0.27.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new RecursiveShapeIterator ptr **new** (const Layout layout, const Cell cell, unsigned int layer, const Region region, bool overlapping = false)

Description: Creates a recursive, single-layer shape iterator with a region.

layout:	The layout which shall be iterated
cell:	The initial cell which shall be iterated (including its children)
layer:	The layer (index) from which the shapes are taken
region:	The search region
overlapping:	If set to true, shapes overlapping the search region are reported, otherwise touching is sufficient

This constructor creates a new recursive shape iterator which delivers the shapes of the given cell plus its children from the layer given by the layer index in the "layer" parameter.

The search is confined to the region given by the "region" parameter. The region needs to be a rectilinear region. If "overlapping" is true, shapes whose bounding box is overlapping the search

region are reported. If "overlapping" is false, shapes whose bounding box touches the search region are reported.

This constructor has been introduced in version 0.25. The 'overlapping' parameter has been made optional in version 0.27.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new RecursiveShapeIterator ptr **new** (const Layout layout, const Cell cell, unsigned int[] layers, const Box box, bool overlapping = false)

Description: Creates a recursive, multi-layer shape iterator with a region.

layout:	The layout which shall be iterated
cell:	The initial cell which shall be iterated (including its children)
layers:	The layer indexes from which the shapes are taken
box:	The search region
overlapping:	If set to true, shapes overlapping the search region are reported, otherwise touching is sufficient

This constructor creates a new recursive shape iterator which delivers the shapes of the given cell plus its children from the layers given by the layer indexes in the "layers" parameter. While iterating use the layer method to retrieve the layer of the current shape.

The search is confined to the region given by the "box" parameter. If "overlapping" is true, shapes whose bounding box is overlapping the search region are reported. If "overlapping" is false, shapes whose bounding box touches the search region are reported.

This constructor has been introduced in version 0.23. The 'overlapping' parameter has been made optional in version 0.27.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new RecursiveShapeIterator ptr **new** (const Layout layout, const Cell cell, unsigned int[] layers, const Region region, bool overlapping = false)

Description: Creates a recursive, multi-layer shape iterator with a region.

layout:	The layout which shall be iterated
cell:	The initial cell which shall be iterated (including its children)
layers:	The layer indexes from which the shapes are taken
region:	The search region
overlapping:	If set to true, shapes overlapping the search region are reported, otherwise touching is sufficient

This constructor creates a new recursive shape iterator which delivers the shapes of the given cell plus its children from the layers given by the layer indexes in the "layers" parameter. While iterating use the layer method to retrieve the layer of the current shape.

The search is confined to the region given by the "region" parameter. The region needs to be a rectilinear region. If "overlapping" is true, shapes whose bounding box is overlapping the search region are reported. If "overlapping" is false, shapes whose bounding box touches the search region are reported.

This constructor has been introduced in version 0.23. The 'overlapping' parameter has been made optional in version 0.27.

Python specific notes:
This method is the default initializer of the object.

next

Signature: void **next**

Description: Increments the iterator

This moves the iterator to the next shape inside the search scope.

overlapping=

Signature: void **overlapping=** (bool flag)

Description: Sets a flag indicating whether overlapping shapes are selected when a region is used

If this flag is false, shapes touching the search region are returned.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'overlapping'. This is the setter.

overlapping?

Signature: *[const]* bool **overlapping?**

Description: Gets a flag indicating whether overlapping shapes are selected when a region is used

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'overlapping'. This is the getter.

path

Signature: *[const]* InstElement[] **path**

Description: Gets the instantiation path of the shape addressed currently

This attribute is a sequence of InstElement objects describing the cell instance path from the initial cell to the current cell containing the current shape.

This method has been introduced in version 0.25.

prop_id

Signature: *[const]* unsigned long **prop_id**

Description: Gets the effective properties ID

The shape iterator supports property filtering and translation. This method will deliver the effective property ID after translation. The original property ID can be obtained from 'shape.prop_id' and is not changed by installing filters or mappers.

prop_id is evaluated by Region objects for example, when they are created from a shape iterator.

See enable_properties, filter_properties, remove_properties and map_properties for details on this feature.

This attribute has been introduced in version 0.28.4.

region

Signature: *[const]* Box **region**

Description: Gets the basic region that this iterator is using

The basic region is the overall box the region iterator iterates over. There may be an additional complex region that confines the region iterator. See complex_region for this attribute.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'region'. This is the getter.

region=

(1) Signature: void **region=** (const Box box_region)

Description: Sets the rectangular region that this iterator is iterating over

See region for a description of this attribute. Setting a simple region will reset the complex region to a rectangle and reset the iterator to the beginning of the sequence. This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'region'. This is the setter.

(2) Signature: void **region=** (const Region complex_region)

Description: Sets the complex region that this iterator is using

See complex_region for a description of this attribute. Setting the complex region will reset the basic region (see region) to the bounding box of the complex region and reset the iterator to the beginning of the sequence.

This method overload has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'region'. This is the setter.

remove_properties

Signature: void **remove_properties**

Description: Removes properties for the given container.

This will remove all properties and prop_id will deliver 0 always (no properties attached). Alternatively you can apply filter_properties or map_properties to enable properties with a specific name key.

Note that property filters/mappers are additive and act in addition (after) the currently installed filter. So effectively after 'remove_properties' you cannot get them back.

This feature has been introduced in version 0.28.4.

reset

Signature: void **reset**

Description: Resets the iterator to the initial state

This method has been introduced in version 0.23.

reset_selection

Signature: void **reset_selection**

Description: Resets the selection to the default state

In the initial state, the top cell and its children are selected. Child cells can be switched on and off together with their sub-hierarchy using select_cells and unselect_cells.

This method will also reset the iterator.

This method has been introduced in version 0.23.

select_all_cells

Signature: void **select_all_cells**

Description: Selects all cells.

This method will set the "selected" mark on all cells. The effect is that subsequent calls of unselect_cells will unselect only the specified cells, not their children, because they are still unselected.

This method will also reset the iterator.

This method has been introduced in version 0.23.

select_cells

(1) Signature: void **select_cells** (unsigned int[] cells)

Description: Unselects the given cells.

This method will sets the "selected" mark on the given cells. That means that these cells or their child cells are visited, unless they are marked as "unselected" again with the unselect_cells method.

The cells are given as a list of cell indexes.

This method will also reset the iterator.

This method has been introduced in version 0.23.

(2) Signature: void **select_cells** (string cells)

Description: Unselects the given cells.

This method will sets the "selected" mark on the given cells. That means that these cells or their child cells are visited, unless they are marked as "unselected" again with the unselect_cells method.

The cells are given as a glob pattern. A glob pattern follows the syntax of file names on the shell (i.e. "A*" are all cells starting with a letter "A").

This method will also reset the iterator.

This method has been introduced in version 0.23.

shape

Signature: *[const]* Shape **shape**

Description: Gets the current shape

Returns the shape currently referred to by the recursive iterator. This shape is not transformed yet and is located in the current cell.

shape_flags

Signature: *[const]* unsigned int **shape_flags**

Description: Gets the shape selection flags

See shape_flags= for a description of that property.

This getter has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'shape_flags'. This is the getter.

shape_flags=

Signature: void **shape_flags=** (unsigned int flags)

Description: Specifies the shape selection flags

The flags are the same then being defined in Shapes (the default is RBA::Shapes::SAll). The flags must be specified before the shapes are being retrieved. Settings the shapes flags will reset the iterator.

Python specific notes:
The object exposes a writable attribute 'shape_flags'. This is the setter.

top_cell

Signature: *[const]* const Cell ptr **top_cell**

Description: Gets the top cell this iterator is connected to

This method has been introduced in version 0.23.

trans

Signature: *[const]* ICplxTrans **trans**

Description: Gets the current transformation by which the shapes must be transformed into the initial cell

The shapes delivered are not transformed. Instead, this transformation must be applied to get the shape in the coordinate system of the top cell.

Starting with version 0.25, this transformation is a int-to-int transformation the 'itrans' method which was providing this transformation before is deprecated.

unselect_all_cells

Signature: void **unselect_all_cells**

Description: Unselects all cells.

This method will set the "unselected" mark on all cells. The effect is that subsequent calls of select_cells will select only the specified cells, not their children, because they are still unselected.

This method will also reset the iterator.

This method has been introduced in version 0.23.

unselect_cells

(1) Signature: void **unselect_cells** (unsigned int[] cells)

Description: Unselects the given cells.

This method will sets the "unselected" mark on the given cells. That means that these cells or their child cells will not be visited, unless they are marked as "selected" again with the select_cells method.

The cells are given as a list of cell indexes.

This method will also reset the iterator.

This method has been introduced in version 0.23.

(2) Signature: void **unselect_cells** (string cells)

Description: Unselects the given cells.

This method will sets the "unselected" mark on the given cells. That means that these cells or their child cells will not be visited, unless they are marked as "selected" again with the select_cells method.

The cells are given as a glob pattern. A glob pattern follows the syntax of file names on the shell (i.e. "A*" are all cells starting with a letter "A").

This method will also reset the iterator.

This method has been introduced in version 0.23.

4.83. API reference - Class PolygonFilter

Notation used in Ruby API documentation

Module: db

Description: A generic polygon filter adaptor

Polygon filters are an efficient way to filter polygons from a Region. To apply a filter, derive your own filter class and pass an instance to the Region#filter or Region#filtered method.

Conceptually, these methods take each polygon from the region and present it to the filter's 'selected' method. Based on the result of this evaluation, the polygon is kept or discarded.

The magic happens when deep mode regions are involved. In that case, the filter will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the filter behaves. You need to configure the filter by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using the filter.

You can skip this step, but the filter algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

Here is some example that filters triangles:

```
class TriangleFilter < RBA::PolygonFilter

  # Constructor
  def initialize
    self.is_isotropic_and_scale_invariant   # the triangle nature is not dependent on the scale or
 orientation
  end

  # Select only triangles
  def selected(polygon)
    return polygon.holes == 0 && polygon.num_points == 3
  end

end

region = ... # some Region
triangles_only = region.filtered(TriangleFilter::new)
```

This class has been introduced in version 0.29.

Public constructors

new PolygonFilter ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.

	void	_unmanage		Marks the object as no longer owned by the script side.
	void	is_isotropic		Indicates that the filter has isotropic properties
	void	is_isotropic_and_scale_invariant		Indicates that the filter is isotropic and scale invariant
	void	is_scale_invariant		Indicates that the filter is scale invariant
	void	requires_raw_input=	(bool flag)	Sets a value indicating whether the filter needs raw (unmerged) input
[const]	bool	requires_raw_input?		Gets a value indicating whether the filter needs raw (unmerged) input
[virtual,const]	bool	selected	(const Polygon polygon)	Selects a polygon
	void	wants_variants=	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	wants_variants?		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) filters are area or perimeter filters. The area or perimeter of a polygon depends on the scale, but not on the orientation of the polygon.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) filter is the square selector. Whether a polygon is a square or not does not depend on the polygon's orientation nor scale.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) filter is the bounding box aspect ratio (height/width) filter. The definition of heigh and width depends on the orientation, but the ratio is independent on scale.

new

Signature: *[static]* new PolygonFilter ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

requires_raw_input=

Signature: void **requires_raw_input=** (bool flag)

Description: Sets a value indicating whether the filter needs raw (unmerged) input

This flag must be set before using this filter. It tells the filter implementation whether the filter wants to have raw input (unmerged). The default value is 'false', meaning that the filter will receive merged polygons ('merged semantics').

Setting this value to false potentially saves some CPU time needed for merging the polygons. Also, raw input means that strange shapes such as dot-like edges, self-overlapping polygons, empty or degenerated polygons are preserved.

Python specific notes:
The object exposes a writable attribute 'requires_raw_input'. This is the setter.

requires_raw_input?

Signature: *[const]* bool **requires_raw_input?**

Description: Gets a value indicating whether the filter needs raw (unmerged) input

See requires_raw_input= for details.

Python specific notes:
The object exposes a readable attribute 'requires_raw_input'. This is the getter.

selected

Signature: *[virtual,const]* bool **selected** (const Polygon polygon)

Description: Selects a polygon

This method is the actual payload. It needs to be reimplemented in a derived class. It needs to analyze the polygon and return 'true' if it should be kept and 'false' if it should be discarded.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.84. API reference - Class PolygonOperator

Notation used in Ruby API documentation

Module: db

Description: A generic polygon operator

Polygon processors are an efficient way to process polygons from a Region. To apply a processor, derive your own operator class and pass an instance to the Region#process or Region#processed method.

Conceptually, these methods take each polygon from the region and present it to the operators' 'process' method. The result of this call is a list of zero to many output polygons derived from the input polygon. The output region is the sum over all these individual results.

The magic happens when deep mode regions are involved. In that case, the processor will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the operator behaves. You need to configure the operator by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using it.

You can skip this step, but the processor algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

Here is some example that shrinks every polygon to half of the size but does not change the position. In this example the 'position' is defined by the center of the bounding box:

```
class ShrinkToHalf < RBA::PolygonOperator

  # Constructor
  def initialize
    self.is_isotropic_and_scale_invariant   # scale or orientation do not matter
  end

  # Shrink to half size
  def process(polygon)
    shift = polygon.bbox.center - RBA::Point::new   # shift vector
    return [ (polygon.moved(-shift) * 0.5).moved(shift) ]
  end

end

region = ... # some Region
shrinked_to_half = region.processed(ShrinkToHalf::new)
```

This class has been introduced in version 0.29.

Public constructors

new PolygonOperator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.

	void	_unmanage		Marks the object as no longer owned by the script side.
	void	is_isotropic		Indicates that the filter has isotropic properties
	void	is_isotropic_and_scale_invariant		Indicates that the filter is isotropic and scale invariant
	void	is_scale_invariant		Indicates that the filter is scale invariant
[virtual,const]	Polygon[]	process	(const Polygon shape)	Processes a shape
	void	requires_raw_input=	(bool flag)	Sets a value indicating whether the processor needs raw (unmerged) input
[const]	bool	requires_raw_input?		Gets a value indicating whether the processor needs raw (unmerged) input
	void	result_is_merged=	(bool flag)	Sets a value indicating whether the processor delivers merged output
[const]	bool	result_is_merged?		Gets a value indicating whether the processor delivers merged output
	void	result_must_not_be_merge	(bool flag)	Sets a value indicating whether the processor's output must not be merged
[const]	bool	result_must_not_be_merged?		Gets a value indicating whether the processor's output must not be merged
	void	wants_variants=	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	wants_variants?		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) processors are size or shrink operators. Size or shrink is not dependent on orientation unless size or shrink needs to be different in x and y direction.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) processor is the convex decomposition operator. The decomposition of a polygon into convex parts is an operation that is not depending on scale nor orientation.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) processor is the rotation operator. Rotation is not depending on scale, but on the original orientation as mirrored versions need to be rotated differently.

new

Signature: *[static]* new PolygonOperator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

process

Signature: *[virtual,const]* Polygon[] **process** (const Polygon shape)

Description: Processes a shape

	This method is the actual payload. It needs to be reimplemented in a derived class. If needs to process the input shape and deliver a list of output shapes. The output list may be empty to entirely discard the input shape. It may also contain more than a single shape. In that case, the number of total shapes may grow during application of the processor.
requires_raw_input=	**Signature**: void **requires_raw_input=** (bool flag) **Description**: Sets a value indicating whether the processor needs raw (unmerged) input This flag must be set before using this processor. It tells the processor implementation whether the processor wants to have raw input (unmerged). The default value is 'false', meaning that the processor will receive merged polygons ('merged semantics'). Setting this value to false potentially saves some CPU time needed for merging the polygons. Also, raw input means that strange shapes such as dot-like edges, self-overlapping polygons, empty or degenerated polygons are preserved. **Python specific notes:** The object exposes a writable attribute 'requires_raw_input'. This is the setter.
requires_raw_input?	**Signature**: *[const]* bool **requires_raw_input?** **Description**: Gets a value indicating whether the processor needs raw (unmerged) input See requires_raw_input= for details. **Python specific notes:** The object exposes a readable attribute 'requires_raw_input'. This is the getter.
result_is_merged=	**Signature**: void **result_is_merged=** (bool flag) **Description**: Sets a value indicating whether the processor delivers merged output This flag must be set before using this processor. If the processor maintains the merged condition by design (output is merged if input is), it is a good idea to set this predicate to 'true'. This will avoid additional merge steps when the resulting collection is used in further operations that need merged input . **Python specific notes:** The object exposes a writable attribute 'result_is_merged'. This is the setter.
result_is_merged?	**Signature**: *[const]* bool **result_is_merged?** **Description**: Gets a value indicating whether the processor delivers merged output See result_is_merged= for details. **Python specific notes:** The object exposes a readable attribute 'result_is_merged'. This is the getter.
result_must_not_be_merged=	**Signature**: void **result_must_not_be_merged=** (bool flag) **Description**: Sets a value indicating whether the processor's output must not be merged This flag must be set before using this processor. The processor can set this flag if it wants to deliver shapes that must not be merged - e.g. point-like edges or strange or degenerated polygons. . **Python specific notes:** The object exposes a writable attribute 'result_must_not_be_merged'. This is the setter.
result_must_not_be_merged	**Signature**: *[const]* bool **result_must_not_be_merged?** **Description**: Gets a value indicating whether the processor's output must not be merged See result_must_not_be_merged= for details. **Python specific notes:**

	The object exposes a readable attribute 'result_must_not_be_merged'. This is the getter.
wants_variants=	**Signature**: void **wants_variants=** (bool flag) **Description**: Sets a value indicating whether the filter prefers cell variants This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false). This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified. **Python specific notes:** The object exposes a writable attribute 'wants_variants'. This is the setter.
wants_variants?	**Signature**: *[const]* bool **wants_variants?** **Description**: Gets a value indicating whether the filter prefers cell variants See wants_variants= for details. **Python specific notes:** The object exposes a readable attribute 'wants_variants'. This is the getter.

4.85. API reference - Class PolygonToEdgeOperator

Notation used in Ruby API documentation

Module: db

Description: A generic polygon-to-edge operator

Polygon processors are an efficient way to process polygons from a Region. To apply a processor, derive your own operator class and pass an instance to the Region#processed method.

Conceptually, these methods take each polygon from the region and present it to the operator's 'process' method. The result of this call is a list of zero to many output edges derived from the input polygon. The output edge collection is the sum over all these individual results.

The magic happens when deep mode regions are involved. In that case, the processor will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the operator behaves. You need to configure the operator by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using it.

You can skip this step, but the processor algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

For a basic example see the PolygonOperator class, with the exception that this incarnation has to deliver edges.

This class has been introduced in version 0.29.

Public constructors

new PolygonToEdgeOperator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**is_isotropic**		Indicates that the filter has isotropic properties
	void	**is_isotropic_and_scale_invariant**		Indicates that the filter is isotropic and scale invariant
	void	**is_scale_invariant**		Indicates that the filter is scale invariant
[virtual,const]	Edge[]	**process**	(const Polygon shape)	Processes a shape
	void	**requires_raw_input=**	(bool flag)	Sets a value indicating whether the processor needs raw (unmerged) input

[const]	bool	**requires_raw_input?**		Gets a value indicating whether the processor needs raw (unmerged) input
	void	**result_is_merged=**	(bool flag)	Sets a value indicating whether the processor delivers merged output
[const]	bool	**result_is_merged?**		Gets a value indicating whether the processor delivers merged output
	void	**result_must_not_be_merge**	(bool flag)	Sets a value indicating whether the processor's output must not be merged
[const]	bool	**result_must_not_be_merged?**		Gets a value indicating whether the processor's output must not be merged
	void	**wants_variants=**	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	**wants_variants?**		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) processors are size or shrink operators. Size or shrink is not dependent on orientation unless size or shrink needs to be different in x and y direction.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) processor is the convex decomposition operator. The decomposition of a polygon into convex parts is an operation that is not depending on scale nor orientation.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) processor is the rotation operator. Rotation is not depending on scale, but on the original orientation as mirrored versions need to be rotated differently.

new

Signature: *[static]* new PolygonToEdgeOperator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

process

Signature: *[virtual,const]* Edge[] **process** (const Polygon shape)

Description: Processes a shape

This method is the actual payload. It needs to be reimplemented in a derived class. If needs to process the input shape and deliver a list of output shapes. The output list may be empty to entirely discard the input shape. It may also contain more than a single shape. In that case, the number of total shapes may grow during application of the processor.

requires_raw_input=

Signature: void **requires_raw_input=** (bool flag)

Description: Sets a value indicating whether the processor needs raw (unmerged) input

This flag must be set before using this processor. It tells the processor implementation whether the processor wants to have raw input (unmerged). The default value is 'false', meaning that the processor will receive merged polygons ('merged semantics').

Setting this value to false potentially saves some CPU time needed for merging the polygons. Also, raw input means that strange shapes such as dot-like edges, self-overlapping polygons, empty or degenerated polygons are preserved.

Python specific notes:
The object exposes a writable attribute 'requires_raw_input'. This is the setter.

requires_raw_input?

Signature: *[const]* bool **requires_raw_input?**

Description: Gets a value indicating whether the processor needs raw (unmerged) input

See requires_raw_input= for details.

Python specific notes:
The object exposes a readable attribute 'requires_raw_input'. This is the getter.

result_is_merged=

Signature: void **result_is_merged=** (bool flag)

Description: Sets a value indicating whether the processor delivers merged output

This flag must be set before using this processor. If the processor maintains the merged condition by design (output is merged if input is), it is a good idea to set this predicate to 'true'. This will avoid additional merge steps when the resulting collection is used in further operations that need merged input .

Python specific notes:
The object exposes a writable attribute 'result_is_merged'. This is the setter.

result_is_merged?

Signature: *[const]* bool **result_is_merged?**

Description: Gets a value indicating whether the processor delivers merged output

See result_is_merged= for details.

Python specific notes:
The object exposes a readable attribute 'result_is_merged'. This is the getter.

result_must_not_be_merged=

Signature: void **result_must_not_be_merged=** (bool flag)

Description: Sets a value indicating whether the processor's output must not be merged

This flag must be set before using this processor. The processor can set this flag if it wants to deliver shapes that must not be merged - e.g. point-like edges or strange or degenerated polygons. .

Python specific notes:
The object exposes a writable attribute 'result_must_not_be_merged'. This is the setter.

result_must_not_be_merged

Signature: *[const]* bool **result_must_not_be_merged?**

Description: Gets a value indicating whether the processor's output must not be merged

See result_must_not_be_merged= for details.

Python specific notes:
The object exposes a readable attribute 'result_must_not_be_merged'. This is the getter.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.86. API reference - Class PolygonToEdgePairOperator

Notation used in Ruby API documentation

Module: db

Description: A generic polygon-to-edge-pair operator

Polygon processors are an efficient way to process polygons from a Region. To apply a processor, derive your own operator class and pass an instance to the Region#processed method.

Conceptually, these methods take each polygon from the region and present it to the operator's 'process' method. The result of this call is a list of zero to many output edge pairs derived from the input polygon. The output edge pair collection is the sum over all these individual results.

The magic happens when deep mode regions are involved. In that case, the processor will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the operator behaves. You need to configure the operator by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using it.

You can skip this step, but the processor algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

For a basic example see the PolygonOperator class, with the exception that this incarnation has to deliver edge pairs.

This class has been introduced in version 0.29.

Public constructors

new PolygonToEdgePairOperator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**is_isotropic**		Indicates that the filter has isotropic properties
	void	**is_isotropic_and_scale_invariant**		Indicates that the filter is isotropic and scale invariant
	void	**is_scale_invariant**		Indicates that the filter is scale invariant
[virtual,const]	EdgePair[]	**process**	(const Polygon shape)	Processes a shape
	void	**requires_raw_input=**	(bool flag)	Sets a value indicating whether the processor needs raw (unmerged) input

[const]	bool	**requires_raw_input?**		Gets a value indicating whether the processor needs raw (unmerged) input
	void	**result_is_merged=**	(bool flag)	Sets a value indicating whether the processor delivers merged output
[const]	bool	**result_is_merged?**		Gets a value indicating whether the processor delivers merged output
	void	**result_must_not_be_merge**	(bool flag)	Sets a value indicating whether the processor's output must not be merged
[const]	bool	**result_must_not_be_merged?**		Gets a value indicating whether the processor's output must not be merged
	void	**wants_variants=**	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	**wants_variants?**		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) processors are size or shrink operators. Size or shrink is not dependent on orientation unless size or shrink needs to be different in x and y direction.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) processor is the convex decomposition operator. The decomposition of a polygon into convex parts is an operation that is not depending on scale nor orientation.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) processor is the rotation operator. Rotation is not depending on scale, but on the original orientation as mirrored versions need to be rotated differently.

new

Signature: *[static]* new PolygonToEdgePairOperator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

process

Signature: *[virtual,const]* EdgePair[] **process** (const Polygon shape)

Description: Processes a shape

This method is the actual payload. It needs to be reimplemented in a derived class. If needs to process the input shape and deliver a list of output shapes. The output list may be empty to entirely discard the input shape. It may also contain more than a single shape. In that case, the number of total shapes may grow during application of the processor.

requires_raw_input=

Signature: void **requires_raw_input=** (bool flag)

Description: Sets a value indicating whether the processor needs raw (unmerged) input

This flag must be set before using this processor. It tells the processor implementation whether the processor wants to have raw input (unmerged). The default value is 'false', meaning that the processor will receive merged polygons ('merged semantics').

Setting this value to false potentially saves some CPU time needed for merging the polygons. Also, raw input means that strange shapes such as dot-like edges, self-overlapping polygons, empty or degenerated polygons are preserved.

Python specific notes:
The object exposes a writable attribute 'requires_raw_input'. This is the setter.

requires_raw_input?

Signature: *[const]* bool **requires_raw_input?**

Description: Gets a value indicating whether the processor needs raw (unmerged) input

See requires_raw_input= for details.

Python specific notes:
The object exposes a readable attribute 'requires_raw_input'. This is the getter.

result_is_merged=

Signature: void **result_is_merged=** (bool flag)

Description: Sets a value indicating whether the processor delivers merged output

This flag must be set before using this processor. If the processor maintains the merged condition by design (output is merged if input is), it is a good idea to set this predicate to 'true'. This will avoid additional merge steps when the resulting collection is used in further operations that need merged input .

Python specific notes:
The object exposes a writable attribute 'result_is_merged'. This is the setter.

result_is_merged?

Signature: *[const]* bool **result_is_merged?**

Description: Gets a value indicating whether the processor delivers merged output

See result_is_merged= for details.

Python specific notes:
The object exposes a readable attribute 'result_is_merged'. This is the getter.

result_must_not_be_merged=

Signature: void **result_must_not_be_merged=** (bool flag)

Description: Sets a value indicating whether the processor's output must not be merged

This flag must be set before using this processor. The processor can set this flag if it wants to deliver shapes that must not be merged - e.g. point-like edges or strange or degenerated polygons. .

Python specific notes:
The object exposes a writable attribute 'result_must_not_be_merged'. This is the setter.

result_must_not_be_merged

Signature: *[const]* bool **result_must_not_be_merged?**

Description: Gets a value indicating whether the processor's output must not be merged

See result_must_not_be_merged= for details.

Python specific notes:
The object exposes a readable attribute 'result_must_not_be_merged'. This is the getter.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.87. API reference - Class Region

Notation used in Ruby API documentation

Module: db

Description: A region (a potentially complex area consisting of multiple polygons)

Class hierarchy: Region » ShapeCollection

Sub-classes: RectFilter, OppositeFilter

This class was introduced to simplify operations on polygon sets like boolean or sizing operations. Regions consist of many polygons and thus are a generalization of single polygons which describes a single coherence set of points. Regions support a variety of operations and have several states.

The region's state can be empty (does not contain anything) or box-like, i.e. the region consists of a single box. In that case, some operations can be simplified. Regions can have merged state. In merged state, regions consist of merged (non-touching, non-self overlapping) polygons. Each polygon describes one coherent area in merged state.

The preferred representation of polygons inside the region are polygons with holes.

Regions are always expressed in database units. If you want to use regions from different database unit domains, scale the regions accordingly, i.e. by using the transformed method.

Regions provide convenient operators for the boolean operations. Hence it is often no longer required to work with the EdgeProcessor class. For example:

```
r1 = RBA::Region::new(RBA::Box::new(0, 0, 100, 100))
r2 = RBA::Region::new(RBA::Box::new(20, 20, 80, 80))
# compute the XOR:
r1_xor_r2 = r1 ^ r2
```

Regions can be used in two different flavors: in raw mode or merged semantics. With merged semantics (the default), connected polygons are considered to belong together and are effectively merged. Overlapping areas are counted once in that mode. Internal edges (i.e. arising from cut lines) are not considered. In raw mode (without merged semantics), each polygon is considered as it is. Overlaps between polygons may exists and merging has to be done explicitly using the merge method. The semantics can be selected using merged_semantics=.

This class has been introduced in version 0.23.

Public constructors

new Region ptr	**new**		Default constructor
new Region ptr	**new**	(Polygon[] array)	Constructor from a polygon array
new Region ptr	**new**	(const Box box)	Box constructor
new Region ptr	**new**	(const Polygon polygon)	Polygon constructor
new Region ptr	**new**	(const SimplePolygon polygon)	Simple polygon constructor
new Region ptr	**new**	(const Path path)	Path constructor
new Region ptr	**new**	(const RecursiveShapeIterator shape_iterator)	Constructor from a hierarchical shape set
new Region ptr	**new**	(const RecursiveShapeIterator shape_iterator, const ICplxTrans trans)	Constructor from a hierarchical shape set with a transformation

new Region ptr	new	(const Shapes shapes)	Constructor from a shapes container
new Region ptr	new	(const Shapes shapes, const ICplxTrans trans)	Constructor from a shapes container with a transformation
new Region ptr	new	(const RecursiveShapeIterator shape_iterator, DeepShapeStore deep_shape_store, double area_ratio = 0, unsigned long max_vertex_count = 0)	Constructor for a deep region from a hierarchical shape set
new Region ptr	new	(const RecursiveShapeIterator shape_iterator, DeepShapeStore deep_shape_store, const ICplxTrans trans, double area_ratio = 0, unsigned long max_vertex_count = 0)	Constructor for a deep region from a hierarchical shape set
new Region ptr	new	(const RecursiveShapeIterator shape_iterator, string expr, bool as_pattern = true, int enl = 1)	Constructor from a text set
new Region ptr	new	(const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, string expr, bool as_pattern = true, int enl = 1)	Constructor from a text set

Public methods

[const]	Region	&	(const Region other)	Returns the boolean AND between self and the other region
	Region	&=	(const Region other)	Performs the boolean AND between self and the other region in-place (modifying self)
[const]	Region	+	(const Region other)	Returns the combined region of self and the other region
	Region	+=	(const Region other)	Adds the polygons of the other region to self
[const]	Region	-	(const Region other)	Returns the boolean NOT between self and the other region
	Region	-=	(const Region other)	Performs the boolean NOT between self and the other region in-place (modifying self)
[const]	const Polygon ptr	[]	(unsigned long n)	Returns the nth polygon of the region
[const]	Region	^	(const Region other)	Returns the boolean XOR between self and the other region

	Region	**^=**	(const Region other)	Performs the boolean XOR between self and the other region in-place (modifying self)
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	Region	**and**	(const Region other, PropertyConstraint property_constraint = IgnoreProperties)	Returns the boolean AND between self and the other region
	Region	**and_with**	(const Region other, PropertyConstraint property_constraint = IgnoreProperties)	Performs the boolean AND between self and the other region in-place (modifying self)
[const]	Region[]	**andnot**	(const Region other, PropertyConstraint property_constraint = IgnoreProperties)	Returns the boolean AND and NOT between self and the other region
[const]	long	**area**		The area of the region
[const]	long	**area**	(const Box rect)	The area of the region (restricted to a rectangle)
	void	**assign**	(const Region other)	Assigns another object to self
[const]	unsigned int	**base_verbosity**		Gets the minimum verbosity for timing reports
	void	**base_verbosity=**	(int verbosity)	Sets the minimum verbosity for timing reports
[const]	Box	**bbox**		Return the bounding box of the region
	void	**break**	(unsigned long max_vertex_count, double max_area_ratio = 0)	Breaks the polygons of the region into smaller ones
	void	**clear**		Clears the region
	variant	**complex_op**	(CompoundRegionOperationNode ptr node,	Executes a complex operation (see CompoundRegionOperationNode for details)

			PropertyConstraint property_constraint = IgnoreProperties)	
[const]	Region	**corners**	(double angle_min = -180, double angle_max = 180, int dim = 1, bool include_min_angle = true, bool include_max_angle = true)	This method will select all corners whose attached edges satisfy the angle condition.
[const]	Edges	**corners_dots**	(double angle_start = -180, double angle_end = 180, bool include_min_angle = true, bool include_max_angle = true)	This method will select all corners whose attached edges satisfy the angle condition.
[const]	EdgePairs	**corners_edge_pairs**	(double angle_start = -180, double angle_end = 180, bool include_min_angle = true, bool include_max_angle = true)	This method will select all corners whose attached edges satisfy the angle condition.
[const]	unsigned long	**count**		Returns the (flat) number of polygons in the region
[const]	Region	**covering**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which are completely covering polygons from the other region
[const]	unsigned long	**data_id**		Returns the data ID (a unique identifier for the underlying data storage)
[const]	new Shapes ptr	**decompose_convex**	(int preferred_orientation = Polygon#PO_any)	Decomposes the region into convex pieces.
[const]	new Region ptr	**decompose_conve**	(int preferred_orientation = Polygon#PO_any)	Decomposes the region into convex pieces into a region.
[const]	new Shapes ptr	**decompose_trapezoids**	(int mode = Polygon#TD_simple)	Decomposes the region into trapezoids.
[const]	new Region ptr	**decompose_trapez**	(int mode = Polygon#TD_simple)	Decomposes the region into trapezoids.
[const]	Region	**delaunay**		Computes a constrained Delaunay triangulation from the given region
[const]	Region	**delaunay**	(double max_area, double min_b = 1)	Computes a refined, constrained Delaunay triangulation from the given region
	void	**disable_progress**		Disable progress reporting
[const]	new Region ptr	**dup**		Creates a copy of self
[const,iter]	Polygon	**each**		Returns each polygon of the region

[const,iter]	Polygon	**each_merged**		Returns each merged polygon of the region
[const]	Edges	**edges**	(EdgeMode mode = All)	Returns an edge collection representing all edges of the polygons in this region
	void	**enable_progress**	(string label)	Enable progress reporting
	void	**enable_properties**		Enables properties for the given container.
[const]	EdgePairs	**enclosed_check**	(const Region other, unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouchi	Performs an inside check with options
[const]	EdgePairs	**enclosing_check**	(const Region other, unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)	Performs an enclosing check with options
[const]	Region	**extents**		Returns a region with the bounding boxes of the polygons
[const]	Region	**extents**	(int d)	Returns a region with the enlarged bounding boxes of the polygons

[const]	Region	**extents**	(int dx, int dy)	Returns a region with the enlarged bounding boxes of the polygons
[const]	void	**fill**	(Cell ptr in_cell, unsigned int fill_cell_index, const Box fc_box, const Point ptr origin = (0, 0), Region ptr remaining_parts = nil, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil, const Box glue_box = ())	A mapping of Cell#fill_region to the Region class
[const]	void	**fill**	(Cell ptr in_cell, unsigned int fill_cell_index, const Box fc_origin, const Vector row_step, const Vector column_step, const Point ptr origin = (0, 0), Region ptr remaining_parts = nil, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil, const Box glue_box = ())	A mapping of Cell#fill_region to the Region class
[const]	void	**fill_multi**	(Cell ptr in_cell, unsigned int fill_cell_index, const Box fc_origin, const Vector row_step, const Vector column_step, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil, const Box glue_box = ())	A mapping of Cell#fill_region to the Region class
	void	**filter**	(const PolygonFilter ptr filter)	Applies a generic filter in place (replacing the polygons from the Region)
	void	**filter_properties**	(variant[] keys)	Filters properties by certain keys.
[const]	Region	**filtered**	(const PolygonFilter ptr filtered)	Applies a generic filter and returns a filtered copy
	Region	**flatten**		Explicitly flattens a region
[const]	EdgePairs	**grid_check**	(int gx, int gy)	Returns a marker for all vertices not being on the given grid
[const]	bool	**has_valid_polygons?**		Returns true if the region is flat and individual polygons can be accessed randomly
[const]	unsigned long	**hier_count**		Returns the (hierarchical) number of polygons in the region
[const]	Region	**holes**		Returns the holes of the region

[const]	Region	**hulls**		Returns the hulls of the region
[const]	Region	**in**	(const Region other)	Returns all polygons which are members of the other region
[const]	Region[]	**in_and_out**	(const Region other)	Returns all polygons which are members and not members of the other region
	void	**insert**	(const Box box)	Inserts a box
	void	**insert**	(const Polygon polygon)	Inserts a polygon
	void	**insert**	(const SimplePolygon polygon)	Inserts a simple polygon
	void	**insert**	(const Path path)	Inserts a path
	void	**insert**	(RecursiveShapeIterator shape_iterator)	Inserts all shapes delivered by the recursive shape iterator into this region
	void	**insert**	(RecursiveShapeIterator shape_iterator, ICplxTrans trans)	Inserts all shapes delivered by the recursive shape iterator into this region with a transformation
	void	**insert**	(Polygon[] array)	Inserts all polygons from the array into this region
	void	**insert**	(const Region region)	Inserts all polygons from the other region into this region
	void	**insert**	(const Shapes shapes)	Inserts all polygons from the shape collection into this region
	void	**insert**	(const Shapes shapes, const Trans trans)	Inserts all polygons from the shape collection into this region with transformation
	void	**insert**	(const Shapes shapes, const ICplxTrans trans)	Inserts all polygons from the shape collection into this region with complex transformation
[const]	void	**insert_into**	(Layout ptr layout, unsigned int cell_index, unsigned int layer)	Inserts this region into the given layout, below the given cell and into the given layer.
[const]	Region	**inside**	(const Region other)	Returns the polygons of this region which are completely inside polygons from the other region
[const]	EdgePairs	**inside_check**	(const Region other, unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter,	Performs an inside check with options

			Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouchi	
[const]	Region	**interacting**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which overlap or touch polygons from the other region
[const]	Region	**interacting**	(const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which overlap or touch edges from the edge collection
[const]	Region	**interacting**	(const Texts other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which overlap or touch texts
[const]	bool	**is_box?**		Returns true, if the region is a simple box
[const]	bool	**is_deep?**		Returns true if the region is a deep (hierarchical) one
[const]	bool	**is_empty?**		Returns true if the region is empty
[const]	bool	**is_merged?**		Returns true if the region is merged
[const]	EdgePairs	**isolated_check**	(unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouchi	Performs a space check between edges of different polygons with options
[const]	Region	**join**	(const Region other)	Returns the combined region of self and the other region
	Region	**join_with**	(const Region other)	Adds the polygons of the other region to self

	void	**map_properties**	(map<variant,variant> key_map)	Maps properties by name key.
[const]	Region	**members_of**	(const Region other)	Returns all polygons which are members of the other region
	Region	**merge**		Merge the region
	Region	**merge**	(int min_wc)	Merge the region with options
	Region	**merge**	(bool min_coherence, int min_wc)	Merge the region with options
[const]	Region	**merged**		Returns the merged region
	Region	**merged**	(int min_wc)	Returns the merged region (with options)
	Region	**merged**	(bool min_coherence, int min_wc)	Returns the merged region (with options)
	void	**merged_semantics=**	(bool f)	Enables or disables merged semantics
[const]	bool	**merged_semantics**		Gets a flag indicating whether merged semantics is enabled
	void	**min_coherence=**	(bool f)	Enable or disable minimum coherence
[const]	bool	**min_coherence?**		Gets a flag indicating whether minimum coherence is selected
[const]	Region	**minkowski_sum**	(const Edge e)	Compute the Minkowski sum of the region and an edge
[const]	Region	**minkowski_sum**	(const Polygon p)	Compute the Minkowski sum of the region and a polygon
[const]	Region	**minkowski_sum**	(const Box b)	Compute the Minkowski sum of the region and a box
[const]	Region	**minkowski_sum**	(Point[] b)	Compute the Minkowski sum of the region and a contour of points (a trace)
	Region	**move**	(const Vector v)	Moves the region
	Region	**move**	(int x, int y)	Moves the region
[const]	Region	**moved**	(const Vector v)	Returns the moved region (does not modify self)
[const]	Region	**moved**	(int x, int y)	Returns the moved region (does not modify self)
[const]	Region	**nets**	(LayoutToNetlist extracted, variant net_prop_name = nil, const Net ptr[] ptr net_filter = nil)	Pulls the net shapes from a LayoutToNetlist database

[const]	Region	**non_rectangles**		Returns all polygons which are not rectangles
[const]	Region	**non_rectilinear**		Returns all polygons which are not rectilinear
[const]	Region	**non_squares**		Returns all polygons which are not squares
[const]	Region	**not**	(const Region other, PropertyConstraint property_constraint = IgnoreProperties)	Returns the boolean NOT between self and the other region
[const]	Region	**not_covering**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which are not completely covering polygons from the other region
[const]	Region	**not_in**	(const Region other)	Returns all polygons which are not members of the other region
[const]	Region	**not_inside**	(const Region other)	Returns the polygons of this region which are not completely inside polygons from the other region
[const]	Region	**not_interacting**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which do not overlap or touch polygons from the other region
[const]	Region	**not_interacting**	(const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which do not overlap or touch edges from the edge collection
[const]	Region	**not_interacting**	(const Texts other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which do not overlap or touch texts
[const]	Region	**not_members_of**	(const Region other)	Returns all polygons which are not members of the other region
[const]	Region	**not_outside**	(const Region other)	Returns the polygons of this region which are not completely outside polygons from the other region
[const]	Region	**not_overlapping**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which do not overlap polygons from the other region
	Region	**not_with**	(const Region other, PropertyConstraint property_constraint = IgnoreProperties)	Performs the boolean NOT between self and the other region in-place (modifying self)

[const]	EdgePairs	**notch_check**	(unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouchi	Performs a space check between edges of the same polygon with options
[const]	Region	**or**	(const Region other)	Returns the boolean OR between self and the other region
	Region	**or_with**	(const Region other)	Performs the boolean OR between self and the other region in-place (modifying self)
[const]	Region	**outside**	(const Region other)	Returns the polygons of this region which are completely outside polygons from the other region
[const]	EdgePairs	**overlap_check**	(const Region other, unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouchi	Performs an overlap check with options
[const]	Region	**overlapping**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which overlap polygons from the other region
[const]	unsigned long	**perimeter**		The total perimeter of the polygons
[const]	unsigned long	**perimeter**	(const Box rect)	The total perimeter of the polygons (restricted to a rectangle)
	void	**process**	(const PolygonOperator ptr process)	Applies a generic polygon processor in place (replacing the polygons from the Region)

[const]	Region	**processed**	(const PolygonOperator ptr processed)	Applies a generic polygon processor and returns a processed copy
[const]	EdgePairs	**processed**	(const PolygonToEdgePairOperator ptr processed)	Applies a generic polygon-to-edge-pair processor and returns an edge pair collection with the results
[const]	Edges	**processed**	(const PolygonToEdgeOperator ptr processed)	Applies a generic polygon-to-edge processor and returns an edge collection with the results
[const]	Region	**pull_inside**	(const Region other)	Returns all polygons of "other" which are inside polygons of this region
[const]	Region	**pull_interacting**	(const Region other)	Returns all polygons of "other" which are interacting with (overlapping, touching) polygons of this region
[const]	Edges	**pull_interacting**	(const Edges other)	Returns all edges of "other" which are interacting with polygons of this region
[const]	Texts	**pull_interacting**	(const Texts other)	Returns all texts of "other" which are interacting with polygons of this region
[const]	Region	**pull_overlapping**	(const Region other)	Returns all polygons of "other" which are overlapping polygons of this region
[const]	double[][]	**rasterize**	(const Point origin, const Vector pixel_size, unsigned int nx, unsigned int ny)	A grayscale rasterizer delivering the area covered per pixel
[const]	double[][]	**rasterize**	(const Point origin, const Vector pixel_distance, const Vector pixel_size, unsigned int nx, unsigned int ny)	A version of 'rasterize' that allows a pixel step distance which is larger than the pixel size
[const]	Region	**rectangles**		Returns all polygons which are rectangles
[const]	Region	**rectilinear**		Returns all polygons which are rectilinear
	void	**remove_properties**		Removes properties for the given container.
	void	**round_corners**	(double r_inner, double r_outer, unsigned int n)	Corner rounding
[const]	Region	**rounded_corners**	(double r_inner, double r_outer, unsigned int n)	Corner rounding
	void	**scale_and_snap**	(int gx, int mx, int dx, int gy, int my,	Scales and snaps the region to the given grid

			int dy)	
[const]	Region	**scaled_and_snapped**	(int gx, int mx, int dx, int gy, int my, int dy)	Returns the scaled and snapped region
	Region	**select_covering**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the polygons of this region which are completely covering polygons from the other region
	Region	**select_inside**	(const Region other)	Selects the polygons of this region which are completely inside polygons from the other region
	Region	**select_interacting**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the polygons from this region which overlap or touch polygons from the other region
	Region	**select_interacting**	(const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the polygons from this region which overlap or touch edges from the edge collection
	Region	**select_interacting**	(const Texts other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the polygons of this region which overlap or touch texts
	Region	**select_not_covering**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the polygons of this region which are not completely covering polygons from the other region
	Region	**select_not_inside**	(const Region other)	Selects the polygons of this region which are not completely inside polygons from the other region
	Region	**select_not_interacting**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the polygons from this region which do not overlap or touch polygons from the other region
	Region	**select_not_interact**	(const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the polygons from this region which do not overlap or touch edges from the edge collection
	Region	**select_not_interacting**	(const Texts other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the polygons of this region which do not overlap or touch texts
	Region	**select_not_outside**	(const Region other)	Selects the polygons of this region which are not completely outside polygons from the other region

	Region	**select_not_overlapping**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the polygons from this region which do not overlap polygons from the other region
	Region	**select_outside**	(const Region other)	Selects the polygons of this region which are completely outside polygons from the other region
	Region	**select_overlapping**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Selects the polygons from this region which overlap polygons from the other region
[const]	EdgePairs	**separation_check**	(const Region other, unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouchi	Performs a separation check with options
	Region	**size**	(int dx, int dy, unsigned int mode)	Anisotropic sizing (biasing)
	Region	**size**	(const Vector dv, unsigned int mode = 2)	Anisotropic sizing (biasing)
	Region	**size**	(int d, unsigned int mode = 2)	Isotropic sizing (biasing)
[const]	Region	**sized**	(int dx, int dy, unsigned int mode)	Returns the anisotropically sized region
[const]	Region	**sized**	(const Vector dv, unsigned int mode = 2)	Returns the (an)isotropically sized region
[const]	Region	**sized**	(int d, unsigned int mode = 2)	Returns the isotropically sized region
	void	**smooth**	(int d, bool keep_hv = false)	Smoothing
[const]	Region	**smoothed**	(int d,	Smoothing

			bool keep_hv = false)	
	void	**snap**	(int gx, int gy)	Snaps the region to the given grid
[const]	Region	**snapped**	(int gx, int gy)	Returns the snapped region
[const]	EdgePairs	**space_check**	(unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)	Performs a space check with options
[const]	Region[]	**split_covering**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which are completely covering polygons from the other region and the ones which are not at the same time
[const]	Region[]	**split_inside**	(const Region other)	Returns the polygons of this region which are completely inside polygons from the other region and the ones which are not at the same time
[const]	Region[]	**split_interacting**	(const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which are interacting with polygons from the other region and the ones which are not at the same time
[const]	Region[]	**split_interacting**	(const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which are interacting with edges from the other edge collection and the ones which are not at the same time
[const]	Region[]	**split_interacting**	(const Texts other, unsigned long min_count = 1, unsigned long max_count = unlimited)	Returns the polygons of this region which are interacting with texts from the other text collection and the ones which are not at the same time
[const]	Region[]	**split_outside**	(const Region other)	Returns the polygons of this region which are completely outside polygons from the other region and the ones which are not at the same time
[const]	Region[]	**split_overlapping**	(const Region other, unsigned long min_count = 1,	Returns the polygons of this region which are overlapping with polygons

			unsigned long max_count = unlimited)	from the other region and the ones which are not at the same time
[const]	Region	**squares**		Returns all polygons which are squares
[const]	Region	**strange_polygon_c**		Returns a region containing those parts of polygons which are "strange"
	void	**strict_handling=**	(bool f)	Enables or disables strict handling
[const]	bool	**strict_handling?**		Gets a flag indicating whether merged semantics is enabled
	void	**swap**	(Region other)	Swap the contents of this region with the contents of another region
[const]	string	**to_s**		Converts the region to a string
[const]	string	**to_s**	(unsigned long max_count)	Converts the region to a string
	Region	**transform**	(const Trans t)	Transform the region (modifies self)
	Region	**transform**	(const ICplxTrans t)	Transform the region with a complex transformation (modifies self)
	Region	**transform**	(const IMatrix2d t)	Transform the region (modifies self)
	Region	**transform**	(const IMatrix3d t)	Transform the region (modifies self)
[const]	Region	**transformed**	(const Trans t)	Transforms the region
[const]	Region	**transformed**	(const ICplxTrans t)	Transforms the region with a complex transformation
[const]	Region	**transformed**	(const IMatrix2d t)	Transforms the region
[const]	Region	**transformed**	(const IMatrix3d t)	Transforms the region
[const]	EdgePairs	**width_check**	(unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouchi	Performs a width check with options
[const]	EdgePairs	**with_angle**	(double angle, bool inverse)	Returns markers on every corner with the given angle (or not with the given angle)

[const]	EdgePairs	**with_angle**	(double amin, double amax, bool inverse)	Returns markers on every corner with an angle of more than amin and less than amax (or the opposite)
[const]	Region	**with_area**	(long area, bool inverse)	Filter the polygons by area
[const]	Region	**with_area**	(variant min_area, variant max_area, bool inverse)	Filter the polygons by area
[const]	Region	**with_area_ratio**	(double ratio, bool inverse)	Filters the polygons by the bounding box area to polygon area ratio
[const]	Region	**with_area_ratio**	(variant min_ratio, variant max_ratio, bool inverse, bool min_included = true, bool max_included = true)	Filters the polygons by the aspect ratio of their bounding boxes
[const]	Region	**with_bbox_aspect_ratio**	(double ratio, bool inverse)	Filters the polygons by the aspect ratio of their bounding boxes
[const]	Region	**with_bbox_aspect_**	(variant min_ratio, variant max_ratio, bool inverse, bool min_included = true, bool max_included = true)	Filters the polygons by the aspect ratio of their bounding boxes
[const]	Region	**with_bbox_height**	(unsigned int height, bool inverse)	Filter the polygons by bounding box height
[const]	Region	**with_bbox_height**	(variant min_height, variant max_height, bool inverse)	Filter the polygons by bounding box height
[const]	Region	**with_bbox_max**	(unsigned int dim, bool inverse)	Filter the polygons by bounding box width or height, whichever is larger
[const]	Region	**with_bbox_max**	(variant min_dim, variant max_dim, bool inverse)	Filter the polygons by bounding box width or height, whichever is larger
[const]	Region	**with_bbox_min**	(unsigned int dim, bool inverse)	Filter the polygons by bounding box width or height, whichever is smaller
[const]	Region	**with_bbox_min**	(variant min_dim, variant max_dim, bool inverse)	Filter the polygons by bounding box width or height, whichever is smaller
[const]	Region	**with_bbox_width**	(unsigned int width, bool inverse)	Filter the polygons by bounding box width
[const]	Region	**with_bbox_width**	(variant min_width, variant max_width, bool inverse)	Filter the polygons by bounding box width
[const]	Region	**with_holes**	(unsigned long nholes, bool inverse)	Filters the polygons by their number of holes

[const]	Region	**with_holes**	(variant min_bholes, variant max_nholes, bool inverse)	Filter the polygons by their number of holes
[const]	Region	**with_perimeter**	(unsigned long perimeter, bool inverse)	Filter the polygons by perimeter
[const]	Region	**with_perimeter**	(variant min_perimeter, variant max_perimeter, bool inverse)	Filter the polygons by perimeter
[const]	Region	**with_relative_height**	(double ratio, bool inverse)	Filters the polygons by the ratio of height to width
[const]	Region	**with_relative_heigh**	(variant min_ratio, variant max_ratio, bool inverse, bool min_included = true, bool max_included = true)	Filters the polygons by the bounding box height to width ratio
[const]	void	**write**	(string filename)	Writes the region to a file
[const]	Region	**xor**	(const Region other)	Returns the boolean XOR between self and the other region
	Region	**xor_with**	(const Region other)	Performs the boolean XOR between self and the other region in-place (modifying self)
[const]	Region	**|**	(const Region other)	Returns the boolean OR between self and the other region
	Region	**|=**	(const Region other)	Performs the boolean OR between self and the other region in-place (modifying self)

Public static methods and constants

[static,const]	EdgeMode	**All**	Selects all edges
[static,const]	EdgeMode	**Concave**	Selects only concave edges
[static,const]	EdgeMode	**Convex**	Selects only convex edges
[static,const]	PropertyConstraint	**DifferentPropertiesConstraint**	Specifies to consider shapes only if their user properties are different
[static,const]	PropertyConstraint	**DifferentPropertiesConst**	Specifies to consider shapes only if their user properties are different
[static,const]	Metrics	**Euclidian**	Specifies Euclidian metrics for the check functions
[static,const]	Region::RectFilter	**FourSidesAllowed**	Allow errors when on all sides
[static,const]	PropertyConstraint	**IgnoreProperties**	Specifies to ignore properties

[static,const]	ZeroDistanceMode	**IncludeZeroDistanceWhe**	Specifies that check functions should include edges when they are collinear and touch
[static,const]	ZeroDistanceMode	**IncludeZeroDistanceWhenOverlapping**	Specifies that check functions should include edges when they overlap
[static,const]	ZeroDistanceMode	**IncludeZeroDistanceWhe**	Specifies that check functions should include edges when they touch
[static,const]	ZeroDistanceMode	**NeverIncludeZeroDistance**	Specifies that check functions should never include edges with zero distance.
[static,const]	Region::OppositeFilter	**NoOppositeFilter**	No opposite filtering
[static,const]	PropertyConstraint	**NoPropertyConstraint**	Specifies not to apply any property constraint
[static,const]	Region::RectFilter	**NoRectFilter**	Specifies no filtering
[static,const]	EdgeMode	**NotConcave**	Selects only edges which are not concave
[static,const]	EdgeMode	**NotConvex**	Selects only edges which are not convex
[static,const]	Region::OppositeFilter	**NotOpposite**	Only errors NOT appearing on opposite sides of a figure will be reported
[static,const]	EdgeMode	**NotStep**	Selects only edges which are not steps
[static,const]	EdgeMode	**NotStepIn**	Selects only edges which are not steps leading inside
[static,const]	EdgeMode	**NotStepOut**	Selects only edges which are not steps leading outside
[static,const]	Region::RectFilter	**OneSideAllowed**	Allow errors on one side
[static,const]	Region::OppositeFilter	**OnlyOpposite**	Only errors appearing on opposite sides of a figure will be reported
[static,const]	Metrics	**Projection**	Specifies projected distance metrics for the check functions
[static,const]	PropertyConstraint	**SamePropertiesConstrai**	Specifies to consider shapes only if their user properties are the same
[static,const]	PropertyConstraint	**SamePropertiesConstraintDrop**	Specifies to consider shapes only if their user properties are the same
[static,const]	Metrics	**Square**	Specifies square metrics for the check functions
[static,const]	EdgeMode	**Step**	Selects only step edges leading inside or outside
[static,const]	EdgeMode	**StepIn**	Selects only step edges leading inside
[static,const]	EdgeMode	**StepOut**	Selects only step edges leading outside

[static,const]	Region::RectFilter	**ThreeSidesAllowed**	Allow errors when on three sides
[static,const]	Region::RectFilter	**TwoConnectedSidesAllowed**	Allow errors on two sides ("L" configuration)
[static,const]	Region::RectFilter	**TwoOppositeSidesAllow**	Allow errors on two opposite sides
[static,const]	Region::RectFilter	**TwoSidesAllowed**	Allow errors on two sides (not specified which)

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	Region	**minkowsky_sum**	(const Edge e)	Use of this method is deprecated. Use minkowski_sum instead
[const]	Region	**minkowsky_sum**	(const Polygon p)	Use of this method is deprecated. Use minkowski_sum instead
[const]	Region	**minkowsky_sum**	(const Box b)	Use of this method is deprecated. Use minkowski_sum instead
[const]	Region	**minkowsky_sum**	(Point[] b)	Use of this method is deprecated. Use minkowski_sum instead
[const]	unsigned long	**size**		Use of this method is deprecated. Use count instead
	Region	**transform_icplx**	(const ICplxTrans t)	Use of this method is deprecated. Use transform instead
[const]	Region	**transformed_icplx**	(const ICplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

&

Signature: *[const]* Region **&** (const Region other)

Description: Returns the boolean AND between self and the other region

Returns: The result of the boolean AND operation

This method will compute the boolean AND (intersection) between two regions. The result is often but not necessarily always merged.

&=

Signature: Region **&=** (const Region other)

Description: Performs the boolean AND between self and the other region in-place (modifying self)

Returns: The region after modification (self)

This method will compute the boolean AND (intersection) between two regions. The result is often but not necessarily always merged.

Note that in Ruby, the '&=' operator actually does not exist, but is emulated by '&' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'and_with' instead.

+

Signature: *[const]* Region **+** (const Region other)

Description: Returns the combined region of self and the other region

Returns: The resulting region

This operator adds the polygons of the other region to self and returns a new combined region. This usually creates unmerged regions and polygons may overlap. Use merge if you want to ensure the result region is merged.

The 'join' alias has been introduced in version 0.28.12.

+=

Signature: Region **+=** (const Region other)

Description: Adds the polygons of the other region to self

Returns: The region after modification (self)

This operator adds the polygons of the other region to self. This usually creates unmerged regions and polygons may overlap. Use merge if you want to ensure the result region is merged.

Note that in Ruby, the '+=' operator actually does not exist, but is emulated by '+' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'join_with' instead.

The 'join_with' alias has been introduced in version 0.28.12.

-

Signature: *[const]* Region **-** (const Region other)

Description: Returns the boolean NOT between self and the other region

Returns: The result of the boolean NOT operation

This method will compute the boolean NOT (intersection) between two regions. The result is often but not necessarily always merged.

-=

Signature: Region **-=** (const Region other)

Description: Performs the boolean NOT between self and the other region in-place (modifying self)

Returns: The region after modification (self)

This method will compute the boolean NOT (intersection) between two regions. The result is often but not necessarily always merged.

Note that in Ruby, the '-=' operator actually does not exist, but is emulated by '-' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'not_with' instead.

All

Signature: *[static,const]* EdgeMode **All**

Description: Selects all edges

Python specific notes:

The object exposes a readable attribute 'All'. This is the getter.

Concave

Signature: *[static,const]* EdgeMode **Concave**

Description: Selects only concave edges

Python specific notes:
The object exposes a readable attribute 'Concave'. This is the getter.

Convex

Signature: *[static,const]* EdgeMode **Convex**

Description: Selects only convex edges

Python specific notes:
The object exposes a readable attribute 'Convex'. This is the getter.

DifferentPropertiesConstraint

Signature: *[static,const]* PropertyConstraint **DifferentPropertiesConstraint**

Description: Specifies to consider shapes only if their user properties are different

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are different. Properties are generated on the output shapes where applicable.

Python specific notes:
The object exposes a readable attribute 'DifferentPropertiesConstraint'. This is the getter.

DifferentPropertiesConstr

Signature: *[static,const]* PropertyConstraint **DifferentPropertiesConstraintDrop**

Description: Specifies to consider shapes only if their user properties are different

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are the same. No properties are generated on the output shapes.

Python specific notes:
The object exposes a readable attribute 'DifferentPropertiesConstraintDrop'. This is the getter.

Euclidian

Signature: *[static,const]* Metrics **Euclidian**

Description: Specifies Euclidian metrics for the check functions

This value can be used for the metrics parameter in the check functions, i.e. width_check. This value specifies Euclidian metrics, i.e. the distance between two points is measured by:

```
d = sqrt(dx^2 + dy^2)
```

All points within a circle with radius d around one point are considered to have a smaller distance than d.

Python specific notes:
The object exposes a readable attribute 'Euclidian'. This is the getter.

FourSidesAllowed

Signature: *[static,const]* Region::RectFilter **FourSidesAllowed**

Description: Allow errors when on all sides

Python specific notes:
The object exposes a readable attribute 'FourSidesAllowed'. This is the getter.

IgnoreProperties

Signature: *[static,const]* PropertyConstraint **IgnoreProperties**

Description: Specifies to ignore properties

When using this constraint - for example on a boolean operation - properties are ignored and are not generated in the output.

Python specific notes:
The object exposes a readable attribute 'IgnoreProperties'. This is the getter.

IncludeZeroDistanceWhe

Signature: *[static,const]* ZeroDistanceMode **IncludeZeroDistanceWhenCollinearAndTouching**

Description: Specifies that check functions should include edges when they are collinear and touch

With this specification, the check functions will also check edges if they share at least one common point and are collinear. This is the mode that includes checking the 'kissing corner' cases when the kissing edges are collinear. This mode was default up to version 0.28.

Python specific notes:
The object exposes a readable attribute 'IncludeZeroDistanceWhenCollinearAndTouching'. This is the getter.

IncludeZeroDistanceWhenOverlapping

Signature: *[static,const]* ZeroDistanceMode **IncludeZeroDistanceWhenOverlapping**

Description: Specifies that check functions should include edges when they overlap

With this specification, the check functions will also check edges which are collinear and share more than a single point. This is the mode that excludes the 'kissing corner' cases.

Python specific notes:
The object exposes a readable attribute 'IncludeZeroDistanceWhenOverlapping'. This is the getter.

IncludeZeroDistanceWhe

Signature: *[static,const]* ZeroDistanceMode **IncludeZeroDistanceWhenTouching**

Description: Specifies that check functions should include edges when they touch

With this specification, the check functions will also check edges if they share at least one common point. This is the mode that includes checking the 'kissing corner' cases. This mode is default for version 0.28.16 and later.

Python specific notes:
The object exposes a readable attribute 'IncludeZeroDistanceWhenTouching'. This is the getter.

NeverIncludeZeroDistance

Signature: *[static,const]* ZeroDistanceMode **NeverIncludeZeroDistance**

Description: Specifies that check functions should never include edges with zero distance.

With this specification, the check functions will ignore edges which are collinear or touch.

Python specific notes:
The object exposes a readable attribute 'NeverIncludeZeroDistance'. This is the getter.

NoOppositeFilter

Signature: *[static,const]* Region::OppositeFilter **NoOppositeFilter**

Description: No opposite filtering

Python specific notes:
The object exposes a readable attribute 'NoOppositeFilter'. This is the getter.

NoPropertyConstraint

Signature: *[static,const]* PropertyConstraint **NoPropertyConstraint**

Description: Specifies not to apply any property constraint

When using this constraint - for example on a boolean operation - shapes are considered regardless of their user properties. Properties are generated on the output shapes where applicable.

Python specific notes:
The object exposes a readable attribute 'NoPropertyConstraint'. This is the getter.

NoRectFilter

Signature: *[static,const]* Region::RectFilter **NoRectFilter**

Description: Specifies no filtering

Python specific notes:
The object exposes a readable attribute 'NoRectFilter'. This is the getter.

NotConcave

Signature: *[static,const]* EdgeMode **NotConcave**

Description: Selects only edges which are not concave

Python specific notes:
The object exposes a readable attribute 'NotConcave'. This is the getter.

NotConvex

Signature: *[static,const]* EdgeMode **NotConvex**

Description: Selects only edges which are not convex

Python specific notes:
The object exposes a readable attribute 'NotConvex'. This is the getter.

NotOpposite

Signature: *[static,const]* Region::OppositeFilter **NotOpposite**

Description: Only errors NOT appearing on opposite sides of a figure will be reported

Python specific notes:
The object exposes a readable attribute 'NotOpposite'. This is the getter.

NotStep

Signature: *[static,const]* EdgeMode **NotStep**

Description: Selects only edges which are not steps

Python specific notes:
The object exposes a readable attribute 'NotStep'. This is the getter.

NotStepIn

Signature: *[static,const]* EdgeMode **NotStepIn**

Description: Selects only edges which are not steps leading inside

Python specific notes:
The object exposes a readable attribute 'NotStepIn'. This is the getter.

NotStepOut

Signature: *[static,const]* EdgeMode **NotStepOut**

Description: Selects only edges which are not steps leading outside

Python specific notes:
The object exposes a readable attribute 'NotStepOut'. This is the getter.

OneSideAllowed

Signature: *[static,const]* Region::RectFilter **OneSideAllowed**

Description: Allow errors on one side

Python specific notes:
The object exposes a readable attribute 'OneSideAllowed'. This is the getter.

OnlyOpposite

Signature: *[static,const]* Region::OppositeFilter **OnlyOpposite**

Description: Only errors appearing on opposite sides of a figure will be reported

Python specific notes:
The object exposes a readable attribute 'OnlyOpposite'. This is the getter.

Projection

Signature: *[static,const]* Metrics **Projection**

Description: Specifies projected distance metrics for the check functions

This value can be used for the metrics parameter in the check functions, i.e. width_check. This value specifies projected metrics, i.e. the distance is defined as the minimum distance measured perpendicular to one edge. That implies that the distance is defined only where two edges have a non-vanishing projection onto each other.

Python specific notes:
The object exposes a readable attribute 'Projection'. This is the getter.

SamePropertiesConstrain

Signature: *[static,const]* PropertyConstraint **SamePropertiesConstraint**

Description: Specifies to consider shapes only if their user properties are the same

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are the same. Properties are generated on the output shapes where applicable.

Python specific notes:
The object exposes a readable attribute 'SamePropertiesConstraint'. This is the getter.

SamePropertiesConstraintDrop

Signature: *[static,const]* PropertyConstraint **SamePropertiesConstraintDrop**

Description: Specifies to consider shapes only if their user properties are the same

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are the same. No properties are generated on the output shapes.

Python specific notes:
The object exposes a readable attribute 'SamePropertiesConstraintDrop'. This is the getter.

Square

Signature: *[static,const]* Metrics **Square**

Description: Specifies square metrics for the check functions

This value can be used for the metrics parameter in the check functions, i.e. width_check. This value specifies square metrics, i.e. the distance between two points is measured by:

```
d = max(abs(dx), abs(dy))
```

All points within a square with length 2*d around one point are considered to have a smaller distance than d in this metrics.

Python specific notes:
The object exposes a readable attribute 'Square'. This is the getter.

Step

Signature: *[static,const]* EdgeMode **Step**

Description: Selects only step edges leading inside or outside

Python specific notes:
The object exposes a readable attribute 'Step'. This is the getter.

StepIn

Signature: *[static,const]* EdgeMode **StepIn**

Description: Selects only step edges leading inside

Python specific notes:
The object exposes a readable attribute 'StepIn'. This is the getter.

StepOut

Signature: *[static,const]* EdgeMode **StepOut**

Description: Selects only step edges leading outside

Python specific notes:
The object exposes a readable attribute 'StepOut'. This is the getter.

ThreeSidesAllowed

Signature: *[static,const]* Region::RectFilter **ThreeSidesAllowed**

Description: Allow errors when on three sides

Python specific notes:
The object exposes a readable attribute 'ThreeSidesAllowed'. This is the getter.

TwoConnectedSidesAllowed

Signature: *[static,const]* Region::RectFilter **TwoConnectedSidesAllowed**

Description: Allow errors on two sides ("L" configuration)

Python specific notes:
The object exposes a readable attribute 'TwoConnectedSidesAllowed'. This is the getter.

TwoOppositeSidesAllowed

Signature: *[static,const]* Region::RectFilter **TwoOppositeSidesAllowed**

Description: Allow errors on two opposite sides

Python specific notes:
The object exposes a readable attribute 'TwoOppositeSidesAllowed'. This is the getter.

TwoSidesAllowed

Signature: *[static,const]* Region::RectFilter **TwoSidesAllowed**

Description: Allow errors on two sides (not specified which)

Python specific notes:
The object exposes a readable attribute 'TwoSidesAllowed'. This is the getter.

[]

Signature: *[const]* const Polygon ptr **[]** (unsigned long n)

Description: Returns the nth polygon of the region

This method returns nil if the index is out of range. It is available for flat regions only - i.e. those for which has_valid_polygons? is true. Use flatten to explicitly flatten a region. This method returns the raw polygon (not merged polygons, even if merged semantics is enabled).

The each iterator is the more general approach to access the polygons.

^

Signature: *[const]* Region **^** (const Region other)

Description: Returns the boolean XOR between self and the other region

Returns: The result of the boolean XOR operation

This method will compute the boolean XOR (intersection) between two regions. The result is often but not necessarily always merged.

The 'xor' alias has been introduced in version 0.28.12.

^=

Signature: Region **^=** (const Region other)

Description: Performs the boolean XOR between self and the other region in-place (modifying self)

Returns: The region after modification (self)

This method will compute the boolean XOR (intersection) between two regions. The result is often but not necessarily always merged.

Note that in Ruby, the '^=' operator actually does not exist, but is emulated by '^' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'xor_with' instead.

The 'xor_with' alias has been introduced in version 0.28.12.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

and

Signature: *[const]* Region **and** (const Region other, PropertyConstraint property_constraint = IgnoreProperties)

Description: Returns the boolean AND between self and the other region

Returns: The result of the boolean AND operation

This method will compute the boolean AND (intersection) between two regions. The result is often but not necessarily always merged. It allows specification of a property constaint - e.g. only performing the boolean operation between shapes with the same user properties.

This variant has been introduced in version 0.28.4.

Python specific notes:
This attribute is available as 'and_' in Python.

and_with

Signature: Region **and_with** (const Region other, PropertyConstraint property_constraint = IgnoreProperties)

Description: Performs the boolean AND between self and the other region in-place (modifying self)

Returns: The region after modification (self)

This method will compute the boolean AND (intersection) between two regions. The result is often but not necessarily always merged. It allows specification of a property constaint - e.g. only performing the boolean operation between shapes with the same user properties.

This variant has been introduced in version 0.28.4.

andnot

Signature: *[const]* Region[] **andnot** (const Region other, PropertyConstraint property_constraint = IgnoreProperties)

Description: Returns the boolean AND and NOT between self and the other region

Returns: A two-element array of regions with the first one being the AND result and the second one being the NOT result

This method will compute the boolean AND and NOT between two regions simultaneously. Because this requires a single sweep only, using this method is faster than doing AND and NOT separately.

This method has been added in version 0.27.

area

(1) Signature: *[const]* long **area**

Description: The area of the region

Merged semantics applies for this method (see merged_semantics= for a description of this concept)
If merged semantics is not enabled, overlapping areas are counted twice.

(2) Signature: *[const]* long **area** (const Box rect)

Description: The area of the region (restricted to a rectangle)

This version will compute the area of the shapes, restricting the computation to the given rectangle.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)
If merged semantics is not enabled, overlapping areas are counted twice.

assign

Signature: void **assign** (const Region other)

Description: Assigns another object to self

base_verbosity

Signature: *[const]* unsigned int **base_verbosity**

Description: Gets the minimum verbosity for timing reports

See base_verbosity= for details.

This method has been introduced in version 0.26.

Python specific notes:
The object exposes a readable attribute 'base_verbosity'. This is the getter.

base_verbosity=

Signature: void **base_verbosity=** (int verbosity)

Description: Sets the minimum verbosity for timing reports

Timing reports will be given only if the verbosity is larger than this value. Detailed reports will be given when the verbosity is more than this value plus 10. In binary operations, the base verbosity of the first argument is considered.

This method has been introduced in version 0.26.

Python specific notes:
The object exposes a writable attribute 'base_verbosity'. This is the setter.

bbox

Signature: *[const]* Box **bbox**

Description: Return the bounding box of the region

The bounding box is the box enclosing all points of all polygons.

break

Signature: void **break** (unsigned long max_vertex_count, double max_area_ratio = 0)

Description: Breaks the polygons of the region into smaller ones

There are two criteria for splitting a polygon: a polygon is split into parts with less then 'max_vertex_count' points and an bounding box-to-polygon area ratio less than 'max_area_ratio'. The area ratio is supposed to render polygons whose bounding box is a better approximation. This applies for example to 'L' shape polygons.

Using a value of 0 for either limit means that the respective limit isn't checked. Breaking happens by cutting the polygons into parts at 'good' locations. The algorithm does not have a specific goal to minimize the number of parts for example. The only goal is to achieve parts within the given limits.

This method has been introduced in version 0.26.

Python specific notes:
This attribute is available as 'break_' in Python.

clear

Signature: void **clear**

Description: Clears the region

complex_op

Signature: variant **complex_op** (CompoundRegionOperationNode ptr node, PropertyConstraint property_constraint = IgnoreProperties)

Description: Executes a complex operation (see CompoundRegionOperationNode for details)

This method has been introduced in version 0.27. The 'property_constraint' parameter controls whether properties are considered: with 'SamePropertiesConstraint' the operation is only applied between shapes with identical properties. With 'DifferentPropertiesConstraint' only between shapes with different properties. This option has been introduced in version 0.28.4.

corners

Signature: *[const]* Region **corners** (double angle_min = -180, double angle_max = 180, int dim = 1, bool include_min_angle = true, bool include_max_angle = true)

Description: This method will select all corners whose attached edges satisfy the angle condition.

The angle values specify a range of angles: all corners whose attached edges form an angle between angle_min and angle_max will be reported boxes with 2*dim x 2*dim dimension. The default dimension is 2x2 DBU.

If 'include_angle_min' is true, the angle condition is >= min. angle, otherwise it is > min. angle. Same for 'include_angle_,ax' and the max. angle.

The angle is measured between the incoming and the outcoming edge in mathematical sense: a positive value is a turn left while a negative value is a turn right. Since polygon contours are oriented clockwise, positive angles will report concave corners while negative ones report convex ones.

A similar function that reports corners as point-like edges is corners_dots.

This method has been introduced in version 0.25. 'include_min_angle' and 'include_max_angle' have been added in version 0.27.

corners_dots

Signature: *[const]* Edges **corners_dots** (double angle_start = -180, double angle_end = 180, bool include_min_angle = true, bool include_max_angle = true)

Description: This method will select all corners whose attached edges satisfy the angle condition.

This method is similar to corners, but delivers an Edges collection with dot-like edges for each corner.

This method has been introduced in version 0.25. 'include_min_angle' and 'include_max_angle' have been added in version 0.27.

corners_edge_pairs

Signature: *[const]* EdgePairs **corners_edge_pairs** (double angle_start = -180, double angle_end = 180, bool include_min_angle = true, bool include_max_angle = true)

Description: This method will select all corners whose attached edges satisfy the angle condition.

This method is similar to corners, but delivers an EdgePairs collection with an edge pairs for each corner. The first edge is the incoming edge of the corner, the second one the outgoing edge.

This method has been introduced in version 0.27.1.

count

Signature: *[const]* unsigned long **count**

Description: Returns the (flat) number of polygons in the region

This returns the number of raw polygons (not merged polygons if merged semantics is enabled). The count is computed 'as if flat', i.e. polygons inside a cell are multiplied by the number of times a cell is instantiated.

The 'count' alias has been provided in version 0.26 to avoid ambiguity with the 'size' method which applies a geometrical bias.

Python specific notes:
This method is also available as 'len(object)'.

covering

Signature: *[const]* Region **covering** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which are completely covering polygons from the other region

Returns: A new region containing the polygons which are covering polygons from the other region

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This attribute is sometimes called 'enclosing' instead of 'covering', but this term is reserved for the respective DRC function.

This method has been introduced in version 0.27.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

data_id

Signature: *[const]* unsigned long **data_id**

Description: Returns the data ID (a unique identifier for the underlying data storage)

This method has been added in version 0.26.

decompose_convex

Signature: *[const]* new Shapes ptr **decompose_convex** (int preferred_orientation = Polygon#PO_any)

Description: Decomposes the region into convex pieces.

This method will return a Shapes container that holds a decomposition of the region into convex, simple polygons. See Polygon#decompose_convex for details. If you want Region output, you should use decompose_convex_to_region.

This method has been introduced in version 0.25.

decompose_convex_to_reg

Signature: *[const]* new Region ptr **decompose_convex_to_region** (int preferred_orientation = Polygon#PO_any)

Description: Decomposes the region into convex pieces into a region.

This method is identical to decompose_convex, but delivers a Region object.

This method has been introduced in version 0.25.

decompose_trapezoids

Signature: *[const]* new Shapes ptr **decompose_trapezoids** (int mode = Polygon#TD_simple)

Description: Decomposes the region into trapezoids.

This method will return a Shapes container that holds a decomposition of the region into trapezoids. See Polygon#decompose_trapezoids for details. If you want Region output, you should use decompose_trapezoids_to_region.

This method has been introduced in version 0.25.

decompose_trapezoids_to_

Signature: *[const]* new Region ptr **decompose_trapezoids_to_region** (int mode = Polygon#TD_simple)

Description: Decomposes the region into trapezoids.

This method is identical to decompose_trapezoids, but delivers a Region object.

This method has been introduced in version 0.25.

delaunay

(1) Signature: *[const]* Region **delaunay**

Description: Computes a constrained Delaunay triangulation from the given region

Returns: A new region holding the triangles of the constrained Delaunay triangulation.

Note that the result is a region in raw mode as otherwise the triangles are likely to get merged later on.

This method has been introduced in version 0.29.

(2) Signature: *[const]* Region **delaunay** (double max_area, double min_b = 1)

Description: Computes a refined, constrained Delaunay triangulation from the given region

Returns: A new region holding the triangles of the refined, constrained Delaunay triangulation.

Refinement is implemented by Chew's second algorithm. A maximum area can be given. Triangles larger than this area will be split. In addition 'skinny' triangles will be resolved where possible. 'skinny'

is defined in terms of shortest edge to circumcircle radius ratio (b). A minimum number for b can be given. The default of 1.0 corresponds to a minimum angle of 30 degree and is usually a good choice. The algorithm is stable up to roughly 1.2 which corresponds to a minimum angle of abouth 37 degree.

The area value is given in terms of DBU units. Picking a value of 0.0 for area and min b will make the implementation skip the refinement step. In that case, the results are identical to the standard constrained Delaunay triangulation.

Note that the result is a region in raw mode as otherwise the triangles are likely to get merged later on.

This method has been introduced in version 0.29.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disable_progress

Signature: void **disable_progress**

Description: Disable progress reporting

Calling this method will disable progress reporting. See enable_progress.

dup

Signature: *[const]* new Region ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each

Signature: *[const,iter]* Polygon **each**

Description: Returns each polygon of the region

This returns the raw polygons (not merged polygons if merged semantics is enabled).

Python specific notes:
This method enables iteration of the object.

each_merged

Signature: *[const,iter]* Polygon **each_merged**

Description: Returns each merged polygon of the region

This returns the raw polygons if merged semantics is disabled or the merged ones if merged semantics is enabled.

edges

Signature: *[const]* Edges **edges** (EdgeMode mode = All)

Description: Returns an edge collection representing all edges of the polygons in this region

This method will decompose the polygons into the individual edges. Edges making up the hulls of the polygons are oriented clockwise while edges making up the holes are oriented counterclockwise.

The 'mode' parameter allows selecting specific edges, such as convex or concave ones. By default, all edges are selected.

The edge collection returned can be manipulated in various ways. See Edges for a description of the features of the edge collection.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

The mode argument has been added in version 0.29.

enable_progress

Signature: void **enable_progress** (string label)

Description: Enable progress reporting

After calling this method, the region will report the progress through a progress bar while expensive operations are running. The label is a text which is put in front of the progress bar. Using a progress bar will imply a performance penalty of a few percent typically.

enable_properties

Signature: void **enable_properties**

Description: Enables properties for the given container.

This method has an effect mainly on original layers and will import properties from such layers. By default, properties are not enabled on original layers. Alternatively you can apply filter_properties or map_properties to enable properties with a specific name key.

This method has been introduced in version 0.28.4.

enclosed_check

Signature: *[const]* EdgePairs **enclosed_check** (const Region other, unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs an inside check with options

d:	The minimum distance for which the polygons are checked
other:	The other region against which to check
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper limit of the projected length of one edge onto another
opposite_filter:	Specifies a filter mode for errors happening on opposite sides of inputs shapes
rect_filter:	Specifies an error filter for rectangular input shapes
negative:	Negative output from the first input
property_constraint:	Specifies whether to consider only shapes with a certain property relation
zero_distance_mode:	Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants.

"ignore_angle" specifies the angle limit of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one limit, pass nil to the respective value.

"shielded" controls whether shielding is applied. Shielding means that rule violations are not detected 'through' other features. Measurements are only made where the opposite edge is unobstructed. Shielding often is not optional as a rule violation in shielded case automatically comes with rule violations between the original and the shielding features. If not necessary, shielding can be disabled by setting this flag to false. In general, this will improve performance somewhat.

"opposite_filter" specifies whether to require or reject errors happening on opposite sides of a figure. "rect_filter" allows suppressing specific error configurations on rectangular input figures.

If "negative" is true, only edges from the first input are output as pseudo edge-pairs where the distance is larger or equal to the limit. This is a way to flag the parts of the first input where the distance to the second input is bigger. Note that only the first input's edges are output. The output is still edge pairs, but each edge pair contains one edge from the original input and the reverse version of the edge as the second edge.

Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

The 'shielded', 'negative', 'not_opposite' and 'rect_sides' options have been introduced in version 0.27. The interpretation of the 'negative' flag has been restriced to first-layout only output in 0.27.1. The 'enclosed_check' alias was introduced in version 0.27.5. 'property_constraint' has been added in version 0.28.4. 'zero_distance_mode' has been added in version 0.29.

enclosing_check

Signature: *[const]* EdgePairs **enclosing_check** (const Region other, unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs an enclosing check with options

d:	The minimum enclosing distance for which the polygons are checked
other:	The other region against which to check
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper limit of the projected length of one edge onto another
opposite_filter:	Specifies a filter mode for errors happening on opposite sides of inputs shapes
rect_filter:	Specifies an error filter for rectangular input shapes
negative:	Negative output from the first input
property_constraint:	Specifies whether to consider only shapes with a certain property relation
zero_distance_mode:	Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants.

"ignore_angle" specifies the angle limit of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one limit, pass nil to the respective value.

"shielded" controls whether shielding is applied. Shielding means that rule violations are not detected 'through' other features. Measurements are only made where the opposite edge is unobstructed. Shielding often is not optional as a rule violation in shielded case automatically comes with rule violations between the original and the shielding features. If not necessary, shielding can be disabled by setting this flag to false. In general, this will improve performance somewhat.

"opposite_filter" specifies whether to require or reject errors happening on opposite sides of a figure. "rect_filter" allows suppressing specific error configurations on rectangular input figures.

If "negative" is true, only edges from the first input are output as pseudo edge-pairs where the enclosure is larger or equal to the limit. This is a way to flag the parts of the first input where the enclosure vs. the second input is bigger. Note that only the first input's edges are output. The output is still edge pairs, but each edge pair contains one edge from the original input and the reverse version of the edge as the second edge.

Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

The 'shielded', 'negative', 'not_opposite' and 'rect_sides' options have been introduced in version 0.27. The interpretation of the 'negative' flag has been restriced to first-layout only output in 0.27.1. 'property_constraint' has been added in version 0.28.4. 'zero_distance_mode' has been added in version 0.29.

extents

(1) Signature: *[const]* Region **extents**

Description: Returns a region with the bounding boxes of the polygons

This method will return a region consisting of the bounding boxes of the polygons. The boxes will not be merged, so it is possible to determine overlaps of these boxes for example.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

(2) Signature: *[const]* Region **extents** (int d)

Description: Returns a region with the enlarged bounding boxes of the polygons

This method will return a region consisting of the bounding boxes of the polygons enlarged by the given distance d. The enlargement is specified per edge, i.e the width and height will be increased by 2*d. The boxes will not be merged, so it is possible to determine overlaps of these boxes for example.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

(3) Signature: *[const]* Region **extents** (int dx, int dy)

Description: Returns a region with the enlarged bounding boxes of the polygons

This method will return a region consisting of the bounding boxes of the polygons enlarged by the given distance dx in x direction and dy in y direction. The enlargement is specified per edge, i.e the width will be increased by 2*dx. The boxes will not be merged, so it is possible to determine overlaps of these boxes for example.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

fill

(1) Signature: *[const]* void **fill** (Cell ptr in_cell, unsigned int fill_cell_index, const Box fc_box, const Point ptr origin = (0, 0), Region ptr remaining_parts = nil, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil, const Box glue_box = ())

Description: A mapping of Cell#fill_region to the Region class

This method is equivalent to Cell#fill_region, but is based on Region (with the cell being the first parameter).

This method has been introduced in version 0.27.

(2) Signature: *[const]* void **fill** (Cell ptr in_cell, unsigned int fill_cell_index, const Box fc_origin, const Vector row_step, const Vector column_step, const Point ptr origin = (0, 0), Region ptr remaining_parts = nil, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil, const Box glue_box = ())

Description: A mapping of Cell#fill_region to the Region class

This method is equivalent to Cell#fill_region, but is based on Region (with the cell being the first parameter).

This method has been introduced in version 0.27.

fill_multi

Signature: *[const]* void **fill_multi** (Cell ptr in_cell, unsigned int fill_cell_index, const Box fc_origin, const Vector row_step, const Vector column_step, const Vector fill_margin = 0,0, Region ptr remaining_polygons = nil, const Box glue_box = ())

Description: A mapping of Cell#fill_region to the Region class

This method is equivalent to Cell#fill_region, but is based on Region (with the cell being the first parameter).

This method has been introduced in version 0.27.

filter

Signature: void **filter** (const PolygonFilter ptr filter)

Description: Applies a generic filter in place (replacing the polygons from the Region)

See PolygonFilter for a description of this feature.

This method has been introduced in version 0.29.

filter_properties

Signature: void **filter_properties** (variant[] keys)

Description: Filters properties by certain keys.

Calling this method on a container will reduce the properties to values with name keys from the 'keys' list. As a side effect, this method enables properties on original layers.

This method has been introduced in version 0.28.4.

filtered

Signature: *[const]* Region **filtered** (const PolygonFilter ptr filtered)

Description: Applies a generic filter and returns a filtered copy

See PolygonFilter for a description of this feature.

This method has been introduced in version 0.29.

flatten

Signature: Region **flatten**

Description: Explicitly flattens a region

If the region is already flat (i.e. has_valid_polygons? returns true), this method will not change it.

Returns 'self', so this method can be used in a dot concatenation.

This method has been introduced in version 0.26.

grid_check

Signature: *[const]* EdgePairs **grid_check** (int gx, int gy)

Description: Returns a marker for all vertices not being on the given grid

This method will return an edge pair object for every vertex whose x coordinate is not a multiple of gx or whose y coordinate is not a multiple of gy. The edge pair objects contain two edges consisting of the same single point - the original vertex.

If gx or gy is 0 or less, the grid is not checked in that direction.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

has_valid_polygons?

Signature: *[const]* bool **has_valid_polygons?**

Description: Returns true if the region is flat and individual polygons can be accessed randomly

This method has been introduced in version 0.26.

hier_count

Signature: *[const]* unsigned long **hier_count**

Description: Returns the (hierarchical) number of polygons in the region

This returns the number of raw polygons (not merged polygons if merged semantics is enabled). The count is computed 'hierarchical', i.e. polygons inside a cell are counted once even if the cell is instantiated multiple times.

This method has been introduced in version 0.27.

holes

Signature: *[const]* Region **holes**

Description: Returns the holes of the region

This method returns all holes as filled polygons.

Merged semantics applies for this method (see merged_semantics= for a description of this concept) If merge semantics is not enabled, the holes may not be detected if the polygons are taken from a hole-less representation (i.e. GDS2 file). Use explicit merge (merge method) in order to merge the polygons and detect holes.

hulls

Signature: *[const]* Region **hulls**

Description: Returns the hulls of the region

This method returns all hulls as polygons. The holes will be removed (filled). Merged semantics applies for this method (see merged_semantics= for a description of this concept) If merge semantics is not enabled, the hull may also enclose holes if the polygons are taken from a hole-less representation (i.e. GDS2 file). Use explicit merge (merge method) in order to merge the polygons and detect holes.

in

Signature: *[const]* Region **in** (const Region other)

Description: Returns all polygons which are members of the other region

This method returns all polygons in self which can be found in the other region as well with exactly the same geometry.

Python specific notes:
This attribute is available as 'in_' in Python.

in_and_out

Signature: *[const]* Region[] **in_and_out** (const Region other)

Description: Returns all polygons which are members and not members of the other region

This method is equivalent to calling members_of and not_members_of, but delivers both results at the same time and is more efficient than two separate calls. The first element returned is the members_of part, the second is the not_members_of part.

This method has been introduced in version 0.28.

insert

(1) Signature: void **insert** (const Box box)

Description: Inserts a box

Inserts a box into the region.

(2) Signature: void **insert** (const Polygon polygon)

Description: Inserts a polygon

Inserts a polygon into the region.

(3) Signature: void **insert** (const SimplePolygon polygon)

Description: Inserts a simple polygon

Inserts a simple polygon into the region.

(4) Signature: void **insert** (const Path path)

Description: Inserts a path

Inserts a path into the region.

(5) Signature: void **insert** (RecursiveShapeIterator shape_iterator)

Description: Inserts all shapes delivered by the recursive shape iterator into this region

This method will insert all shapes delivered by the shape iterator and insert them into the region. Text objects and edges are not inserted, because they cannot be converted to polygons.

(6) Signature: void **insert** (RecursiveShapeIterator shape_iterator, ICplxTrans trans)

Description: Inserts all shapes delivered by the recursive shape iterator into this region with a transformation

This method will insert all shapes delivered by the shape iterator and insert them into the region. Text objects and edges are not inserted, because they cannot be converted to polygons. This variant will apply the given transformation to the shapes. This is useful to scale the shapes to a specific database unit for example.

(7) Signature: void **insert** (Polygon[] array)

Description: Inserts all polygons from the array into this region

(8) Signature: void **insert** (const Region region)

Description: Inserts all polygons from the other region into this region

This method has been introduced in version 0.25.

(9) Signature: void **insert** (const Shapes shapes)

Description: Inserts all polygons from the shape collection into this region

This method takes each "polygon-like" shape from the shape collection and inserts this shape into the region. Paths and boxes are converted to polygons during this process. Edges and text objects are ignored.

This method has been introduced in version 0.25.

(10) Signature: void **insert** (const Shapes shapes, const Trans trans)

Description: Inserts all polygons from the shape collection into this region with transformation

This method takes each "polygon-like" shape from the shape collection and inserts this shape into the region after applying the given transformation. Paths and boxes are converted to polygons during this process. Edges and text objects are ignored.

This method has been introduced in version 0.25.

(11) Signature: void **insert** (const Shapes shapes, const ICplxTrans trans)

Description: Inserts all polygons from the shape collection into this region with complex transformation

This method takes each "polygon-like" shape from the shape collection and inserts this shape into the region after applying the given complex transformation. Paths and boxes are converted to polygons during this process. Edges and text objects are ignored.

This method has been introduced in version 0.25.

insert_into

Signature: *[const]* void **insert_into** (Layout ptr layout, unsigned int cell_index, unsigned int layer)

Description: Inserts this region into the given layout, below the given cell and into the given layer.

If the region is a hierarchical one, a suitable hierarchy will be built below the top cell or and existing hierarchy will be reused.

This method has been introduced in version 0.26.

inside

Signature: *[const]* Region **inside** (const Region other)

Description: Returns the polygons of this region which are completely inside polygons from the other region

Returns:	A new region containing the polygons which are inside polygons from the other region

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

inside_check

Signature: *[const]* EdgePairs **inside_check** (const Region other, unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs an inside check with options

d:	The minimum distance for which the polygons are checked
other:	The other region against which to check
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another

max_projection:	The upper limit of the projected length of one edge onto another
opposite_filter:	Specifies a filter mode for errors happening on opposite sides of inputs shapes
rect_filter:	Specifies an error filter for rectangular input shapes
negative:	Negative output from the first input
property_constraint:	Specifies whether to consider only shapes with a certain property relation
zero_distance_mode:	Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants.

"ignore_angle" specifies the angle limit of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one limit, pass nil to the respective value.

"shielded" controls whether shielding is applied. Shielding means that rule violations are not detected 'through' other features. Measurements are only made where the opposite edge is unobstructed. Shielding often is not optional as a rule violation in shielded case automatically comes with rule violations between the original and the shielding features. If not necessary, shielding can be disabled by setting this flag to false. In general, this will improve performance somewhat.

"opposite_filter" specifies whether to require or reject errors happening on opposite sides of a figure. "rect_filter" allows suppressing specific error configurations on rectangular input figures.

If "negative" is true, only edges from the first input are output as pseudo edge-pairs where the distance is larger or equal to the limit. This is a way to flag the parts of the first input where the distance to the second input is bigger. Note that only the first input's edges are output. The output is still edge pairs, but each edge pair contains one edge from the original input and the reverse version of the edge as the second edge.

Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

The 'shielded', 'negative', 'not_opposite' and 'rect_sides' options have been introduced in version 0.27. The interpretation of the 'negative' flag has been restriced to first-layout only output in 0.27.1. The 'enclosed_check' alias was introduced in version 0.27.5. 'property_constraint' has been added in version 0.28.4. 'zero_distance_mode' has been added in version 0.29.

interacting

(1) Signature: *[const]* Region **interacting** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which overlap or touch polygons from the other region

Returns:	A new region containing the polygons overlapping or touching polygons from the other region

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with (different) polygons of the other region to make the polygon selected. A polygon is selected by this method if the number of polygons interacting with a polygon of this region is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

The min_count and max_count arguments have been added in version 0.27.

(2) Signature: *[const]* Region **interacting** (const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which overlap or touch edges from the edge collection

Returns: A new region containing the polygons overlapping or touching edges from the edge collection

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with edges of the edge collection to make the polygon selected. A polygon is selected by this method if the number of edges interacting with the polygon is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.25. The min_count and max_count arguments have been added in version 0.27.

(3) Signature: *[const]* Region **interacting** (const Texts other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which overlap or touch texts

Returns: A new region containing the polygons overlapping or touching texts

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with texts of the text collection to make the polygon selected. A polygon is selected by this method if the number of texts interacting with the polygon is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27

is_box?

Signature: *[const]* bool **is_box?**

Description: Returns true, if the region is a simple box

Returns: True if the region is a box.

This method does not apply implicit merging if merge semantics is enabled. If the region is not merged, this method may return false even if the merged region would be a box.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_deep?

Signature: *[const]* bool **is_deep?**

Description: Returns true if the region is a deep (hierarchical) one

This method has been added in version 0.26.

is_empty?

Signature: *[const]* bool **is_empty?**

Description: Returns true if the region is empty

is_merged?

Signature: *[const]* bool **is_merged?**

Description: Returns true if the region is merged

If the region is merged, polygons will not touch or overlap. You can ensure merged state by calling merge.

isolated_check

Signature: *[const]* EdgePairs **isolated_check** (unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs a space check between edges of different polygons with options

d:	The minimum space for which the polygons are checked
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper limit of the projected length of one edge onto another
opposite_filter:	Specifies a filter mode for errors happening on opposite sides of inputs shapes
rect_filter:	Specifies an error filter for rectangular input shapes
negative:	If true, edges not violation the condition will be output as pseudo-edge pairs
property_constraint:	Specifies whether to consider only shapes with a certain property relation
zero_distance_mode:	Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants.

"ignore_angle" specifies the angle limit of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one limit, pass nil to the respective value.

"shielded" controls whether shielding is applied. Shielding means that rule violations are not detected 'through' other features. Measurements are only made where the opposite edge is unobstructed. Shielding often is not optional as a rule violation in shielded case automatically comes with rule violations between the original and the shielding features. If not necessary, shielding can be disabled by setting this flag to false. In general, this will improve performance somewhat.

"opposite_filter" specifies whether to require or reject errors happening on opposite sides of a figure. "rect_filter" allows suppressing specific error configurations on rectangular input figures.

Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

The 'shielded', 'negative', 'not_opposite' and 'rect_sides' options have been introduced in version 0.27. 'property_constraint' has been added in version 0.28.4. 'zero_distance_mode' has been added in version 0.29.

join

Signature: *[const]* Region **join** (const Region other)

Description: Returns the combined region of self and the other region

Returns: The resulting region

This operator adds the polygons of the other region to self and returns a new combined region. This usually creates unmerged regions and polygons may overlap. Use merge if you want to ensure the result region is merged.

The 'join' alias has been introduced in version 0.28.12.

join_with

Signature: Region **join_with** (const Region other)

Description: Adds the polygons of the other region to self

Returns: The region after modification (self)

This operator adds the polygons of the other region to self. This usually creates unmerged regions and polygons may overlap. Use merge if you want to ensure the result region is merged.

Note that in Ruby, the '+=' operator actually does not exist, but is emulated by '+' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'join_with' instead.

The 'join_with' alias has been introduced in version 0.28.12.

map_properties

Signature: void **map_properties** (map<variant,variant> key_map)

Description: Maps properties by name key.

Calling this method on a container will reduce the properties to values with name keys from the 'keys' hash and renames the properties. Properties not listed in the key map will be removed. As a side effect, this method enables properties on original layers.

This method has been introduced in version 0.28.4.

members_of

Signature: *[const]* Region **members_of** (const Region other)

Description: Returns all polygons which are members of the other region

This method returns all polygons in self which can be found in the other region as well with exactly the same geometry.

Python specific notes:
This attribute is available as 'in_' in Python.

merge

(1) Signature: Region **merge**

Description: Merge the region

Returns: The region after is has been merged (self).

Merging removes overlaps and joins touching polygons. If the region is already merged, this method does nothing

(2) Signature: Region **merge** (int min_wc)

Description: Merge the region with options

min_wc: Overlap selection

Returns: The region after is has been merged (self).

Merging removes overlaps and joins touching polygons. This version provides one additional option: "min_wc" controls whether output is only produced if multiple polygons overlap. The value specifies the number of polygons that need to overlap. A value of 2 means that output is only produced if two or more polygons overlap.

This method is equivalent to "merge(false, min_wc).

(3) Signature: Region **merge** (bool min_coherence, int min_wc)

Description: Merge the region with options

min_coherence:	A flag indicating whether the resulting polygons shall have minimum coherence
min_wc:	Overlap selection
Returns:	The region after is has been merged (self).

Merging removes overlaps and joins touching polygons. This version provides two additional options: if "min_coherence" is set to true, "kissing corners" are resolved by producing separate polygons. "min_wc" controls whether output is only produced if multiple polygons overlap. The value specifies the number of polygons that need to overlap. A value of 2 means that output is only produced if two or more polygons overlap.

merged

(1) Signature: *[const]* Region **merged**

Description: Returns the merged region

Returns:	The region after is has been merged.

Merging removes overlaps and joins touching polygons. If the region is already merged, this method does nothing. In contrast to merge, this method does not modify the region but returns a merged copy.

(2) Signature: Region **merged** (int min_wc)

Description: Returns the merged region (with options)

Returns:	The region after is has been merged.

This version provides one additional options: "min_wc" controls whether output is only produced if multiple polygons overlap. The value specifies the number of polygons that need to overlap. A value of 2 means that output is only produced if two or more polygons overlap.

This method is equivalent to "merged(false, min_wc)".

In contrast to merge, this method does not modify the region but returns a merged copy.

(3) Signature: Region **merged** (bool min_coherence, int min_wc)

Description: Returns the merged region (with options)

min_coherence:	A flag indicating whether the resulting polygons shall have minimum coherence
min_wc:	Overlap selection
Returns:	The region after is has been merged (self).

Merging removes overlaps and joins touching polygons. This version provides two additional options: if "min_coherence" is set to true, "kissing corners" are resolved by producing separate polygons. "min_wc" controls whether output is only produced if multiple polygons overlap. The value specifies the number of polygons that need to overlap. A value of 2 means that output is only produced if two or more polygons overlap.

In contrast to merge, this method does not modify the region but returns a merged copy.

merged_semantics=

Signature: void **merged_semantics=** (bool f)

Description: Enables or disables merged semantics

If merged semantics is enabled (the default), coherent polygons will be considered as single regions and artificial edges such as cut-lines will not be considered. Merged semantics thus is equivalent to considering coherent areas rather than single polygons

Python specific notes:
The object exposes a writable attribute 'merged_semantics'. This is the setter.

merged_semantics?

Signature: *[const]* bool **merged_semantics?**

Description: Gets a flag indicating whether merged semantics is enabled

See merged_semantics= for a description of this attribute.

Python specific notes:
The object exposes a readable attribute 'merged_semantics'. This is the getter.

min_coherence=

Signature: void **min_coherence=** (bool f)

Description: Enable or disable minimum coherence

If minimum coherence is set, the merge operations (explicit merge with merge or implicit merge through merged_semantics) are performed using minimum coherence mode. The coherence mode determines how kissing-corner situations are resolved. If minimum coherence is selected, they are resolved such that multiple polygons are created which touch at a corner).

The default setting is maximum coherence (min_coherence = false).

Python specific notes:
The object exposes a writable attribute 'min_coherence'. This is the setter.

min_coherence?

Signature: *[const]* bool **min_coherence?**

Description: Gets a flag indicating whether minimum coherence is selected

See min_coherence= for a description of this attribute.

Python specific notes:
The object exposes a readable attribute 'min_coherence'. This is the getter.

minkowski_sum

(1) Signature: *[const]* Region **minkowski_sum** (const Edge e)

Description: Compute the Minkowski sum of the region and an edge

e:	The edge.
Returns:	The new polygons representing the Minkowski sum with the edge e.

The Minkowski sum of a region and an edge basically results in the area covered when "dragging" the region along the line given by the edge. The effect is similar to drawing the line with a pencil that has the shape of the given region.

The resulting polygons are not merged. In order to remove overlaps, use the merge or merged method.Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

(2) Signature: *[const]* Region **minkowski_sum** (const Polygon p)

Description: Compute the Minkowski sum of the region and a polygon

p:	The first argument.
Returns:	The new polygons representing the Minkowski sum of self and p.

The Minkowski sum of a region and a polygon is basically the result of "painting" the region with a pen that has the shape of the second polygon.

The resulting polygons are not merged. In order to remove overlaps, use the merge or merged method.Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

(3) Signature: *[const]* Region **minkowski_sum** (const Box b)

Description: Compute the Minkowski sum of the region and a box

b:	The box.
Returns:	The new polygons representing the Minkowski sum of self and the box.

The result is equivalent to the region-with-polygon Minkowski sum with the box used as the second polygon.

The resulting polygons are not merged. In order to remove overlaps, use the merge or merged method.Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

(4) Signature: *[const]* Region **minkowski_sum** (Point[] b)

Description: Compute the Minkowski sum of the region and a contour of points (a trace)

b:	The contour (a series of points forming the trace).
Returns:	The new polygons representing the Minkowski sum of self and the contour.

The Minkowski sum of a region and a contour basically results in the area covered when "dragging" the region along the contour. The effect is similar to drawing the contour with a pencil that has the shape of the given region.

The resulting polygons are not merged. In order to remove overlaps, use the merge or merged method.Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

minkowsky_sum

(1) Signature: *[const]* Region **minkowsky_sum** (const Edge e)

Description: Compute the Minkowski sum of the region and an edge

e:	The edge.
Returns:	The new polygons representing the Minkowski sum with the edge e.

Use of this method is deprecated. Use minkowski_sum instead

The Minkowski sum of a region and an edge basically results in the area covered when "dragging" the region along the line given by the edge. The effect is similar to drawing the line with a pencil that has the shape of the given region.

The resulting polygons are not merged. In order to remove overlaps, use the merge or merged method.Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

(2) Signature: *[const]* Region **minkowsky_sum** (const Polygon p)

Description: Compute the Minkowski sum of the region and a polygon

p:	The first argument.
Returns:	The new polygons representing the Minkowski sum of self and p.

Use of this method is deprecated. Use minkowski_sum instead

The Minkowski sum of a region and a polygon is basically the result of "painting" the region with a pen that has the shape of the second polygon.

The resulting polygons are not merged. In order to remove overlaps, use the merge or merged method.Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

(3) Signature: *[const]* Region **minkowsky_sum** (const Box b)

Description: Compute the Minkowski sum of the region and a box

b:	The box.
Returns:	The new polygons representing the Minkowski sum of self and the box.

Use of this method is deprecated. Use minkowski_sum instead

The result is equivalent to the region-with-polygon Minkowski sum with the box used as the second polygon.

The resulting polygons are not merged. In order to remove overlaps, use the merge or merged method.Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

(4) Signature: *[const]* Region **minkowsky_sum** (Point[] b)

Description: Compute the Minkowski sum of the region and a contour of points (a trace)

b:	The contour (a series of points forming the trace).
Returns:	The new polygons representing the Minkowski sum of self and the contour.

Use of this method is deprecated. Use minkowski_sum instead

The Minkowski sum of a region and a contour basically results in the area covered when "dragging" the region along the contour. The effect is similar to drawing the contour with a pencil that has the shape of the given region.

The resulting polygons are not merged. In order to remove overlaps, use the merge or merged method.Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

move

(1) Signature: Region **move** (const Vector v)

Description: Moves the region

v:	The distance to move the region.
Returns:	The moved region (self).

Moves the polygon by the given offset and returns the moved region. The region is overwritten.

Starting with version 0.25 this method accepts a vector argument.

(2) Signature: Region **move** (int x, int y)

Description: Moves the region

x:	The x distance to move the region.
y:	The y distance to move the region.
Returns:	The moved region (self).

Moves the region by the given offset and returns the moved region. The region is overwritten.

moved

(1) Signature: *[const]* Region **moved** (const Vector v)

Description: Returns the moved region (does not modify self)

p:	The distance to move the region.
Returns:	The moved region.

Moves the region by the given offset and returns the moved region. The region is not modified.

Starting with version 0.25 this method accepts a vector argument.

(2) Signature: *[const]* Region **moved** (int x, int y)

Description: Returns the moved region (does not modify self)

x:	The x distance to move the region.
y:	The y distance to move the region.
Returns:	The moved region.

Moves the region by the given offset and returns the moved region. The region is not modified.

nets

Signature: *[const]* Region **nets** (LayoutToNetlist extracted, variant net_prop_name = nil, const Net ptr[] ptr net_filter = nil)

Description: Pulls the net shapes from a LayoutToNetlist database

This method will create a new layer with the net shapes from the LayoutToNetlist database, provided that this region was an input to the netlist extraction on this database.

A (circuit name, net name) tuple will be attached as properties to the shapes if 'net_prop_name' is given and not nil. This allows generating unique properties per shape, flagging the net the shape is on. This feature is good for performing net-dependent booleans and DRC checks.

A net filter can be provided with the 'net_filter' argument. If given, only nets from this set are produced. Example:

```
connect(metal1, via1)
connect(via1, metal2)

metal1_all_nets = metal1.nets
```

This method was introduced in version 0.28.4

new

(1) Signature: *[static]* new Region ptr **new**

Description: Default constructor

This constructor creates an empty region.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Region ptr **new** (Polygon[] array)

Description: Constructor from a polygon array

This constructor creates a region from an array of polygons.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Region ptr **new** (const Box box)

Description: Box constructor

This constructor creates a region from a box.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Region ptr **new** (const Polygon polygon)

Description: Polygon constructor

This constructor creates a region from a polygon.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new Region ptr **new** (const SimplePolygon polygon)

Description: Simple polygon constructor

This constructor creates a region from a simple polygon.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new Region ptr **new** (const Path path)

Description: Path constructor

This constructor creates a region from a path.

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new Region ptr **new** (const RecursiveShapeIterator shape_iterator)

Description: Constructor from a hierarchical shape set

This constructor creates a region from the shapes delivered by the given recursive shape iterator. Text objects and edges are not inserted, because they cannot be converted to polygons. This method allows feeding the shapes from a hierarchy of cells into the region.

```
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
r = RBA::Region::new(layout.begin_shapes(cell, layer))
```

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new Region ptr **new** (const RecursiveShapeIterator shape_iterator, const ICplxTrans trans)

Description: Constructor from a hierarchical shape set with a transformation

This constructor creates a region from the shapes delivered by the given recursive shape iterator. Text objects and edges are not inserted, because they cannot be converted to polygons. On the delivered shapes it applies the given transformation. This method allows feeding the shapes from a hierarchy of cells into the region. The transformation is useful to scale to a specific database unit for example.

```
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
dbu    = 0.1 # the target database unit
r = RBA::Region::new(layout.begin_shapes(cell, layer),
 RBA::ICplxTrans::new(layout.dbu / dbu))
```

Python specific notes:
This method is the default initializer of the object.

(9) Signature: *[static]* new Region ptr **new** (const Shapes shapes)

Description: Constructor from a shapes container

This constructor creates a region from the shapes container. Text objects and edges are not inserted, because they cannot be converted to polygons. This method allows feeding the shapes from a hierarchy of cells into the region.

This constructor has been introduced in version 0.25 and extended in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(10) Signature: *[static]* new Region ptr **new** (const Shapes shapes, const ICplxTrans trans)

Description: Constructor from a shapes container with a transformation

This constructor creates a region from the shapes container after applying the transformation. Text objects and edges are not inserted, because they cannot be converted to polygons. This method allows feeding the shapes from a hierarchy of cells into the region.

This constructor variant has been introduced in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(11) Signature: *[static]* new Region ptr **new** (const RecursiveShapeIterator shape_iterator, DeepShapeStore deep_shape_store, double area_ratio = 0, unsigned long max_vertex_count = 0)

Description: Constructor for a deep region from a hierarchical shape set

shape_iterator:	The recursive shape iterator which delivers the hierarchy to take
deep_shape_store:	The hierarchical heap (see there)
area_ratio:	The maximum ratio of bounding box to polygon area before polygons are split

This constructor creates a hierarchical region. Use a DeepShapeStore object to supply the hierarchical heap. See DeepShapeStore for more details.

'area_ratio' and 'max_vertex' supply two optimization parameters which control how big polygons are split to reduce the region's polygon complexity.

This method has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

(12) Signature: *[static]* new Region ptr **new** (const RecursiveShapeIterator shape_iterator, DeepShapeStore deep_shape_store, const ICplxTrans trans, double area_ratio = 0, unsigned long max_vertex_count = 0)

Description: Constructor for a deep region from a hierarchical shape set

shape_iterator:	The recursive shape iterator which delivers the hierarchy to take
deep_shape_store:	The hierarchical heap (see there)
area_ratio:	The maximum ratio of bounding box to polygon area before polygons are split
trans:	The transformation to apply when storing the layout data

This constructor creates a hierarchical region. Use a DeepShapeStore object to supply the hierarchical heap. See DeepShapeStore for more details.

'area_ratio' and 'max_vertex' supply two optimization parameters which control how big polygons are split to reduce the region's polygon complexity.

The transformation is useful to scale to a specific database unit for example.

This method has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

(13) Signature: *[static]* new Region ptr **new** (const RecursiveShapeIterator shape_iterator, string expr, bool as_pattern = true, int enl = 1)

Description: Constructor from a text set

shape_iterator:	The iterator from which to derive the texts
expr:	The selection string
as_pattern:	If true, the selection string is treated as a glob pattern. Otherwise the match is exact.
enl:	The per-side enlargement of the box to mark the text (1 gives a 2x2 DBU box)

This special constructor will create a region from the text objects delivered by the shape iterator. Each text object will give a small (non-empty) box that represents the text origin. Texts can be selected by their strings - either through a glob pattern or by exact comparison with the given string. The following options are available:

```
region = RBA::Region::new(iter, "*")           # all texts
region = RBA::Region::new(iter, "A*")          # all texts starting with an 'A'
region = RBA::Region::new(iter, "A*", false)   # all texts exactly matching
 'A*'
```

This method has been introduced in version 0.25. The enlargement parameter has been added in version 0.26.

Python specific notes:
This method is the default initializer of the object.

(14) Signature: *[static]* new Region ptr **new** (const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, string expr, bool as_pattern = true, int enl = 1)

Description: Constructor from a text set

shape_iterator:	The iterator from which to derive the texts
dss:	The DeepShapeStore object that acts as a heap for hierarchical operations.
expr:	The selection string
as_pattern:	If true, the selection string is treated as a glob pattern. Otherwise the match is exact.
enl:	The per-side enlargement of the box to mark the text (1 gives a 2x2 DBU box)

This special constructor will create a deep region from the text objects delivered by the shape iterator. Each text object will give a small (non-empty) box that represents the text origin. Texts can be selected by their strings - either through a glob pattern or by exact comparison with the given string. The following options are available:

```
region = RBA::Region::new(iter, dss, "*")          # all texts
region = RBA::Region::new(iter, dss, "A*")         # all texts starting with
 an 'A'
region = RBA::Region::new(iter, dss, "A*", false)  # all texts exactly
 matching 'A*'
```

This variant has been introduced in version 0.26.

Python specific notes:
This method is the default initializer of the object.

non_rectangles

Signature: *[const]* Region **non_rectangles**

Description: Returns all polygons which are not rectangles

This method returns all polygons in self which are not rectangles.Merged semantics applies for this method (see merged_semantics= for a description of this concept)

non_rectilinear

Signature: *[const]* Region **non_rectilinear**

Description: Returns all polygons which are not rectilinear

This method returns all polygons in self which are not rectilinear.Merged semantics applies for this method (see merged_semantics= for a description of this concept)

non_squares

Signature: *[const]* Region **non_squares**

Description: Returns all polygons which are not squares

This method returns all polygons in self which are not squares.Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27.

not

Signature: *[const]* Region **not** (const Region other, PropertyConstraint property_constraint = IgnoreProperties)

Description: Returns the boolean NOT between self and the other region

Returns: The result of the boolean NOT operation

This method will compute the boolean NOT (intersection) between two regions. The result is often but not necessarily always merged. It allows specification of a property constaint - e.g. only performing the boolean operation between shapes with the same user properties.

This variant has been introduced in version 0.28.4.

Python specific notes:
This attribute is available as 'not_' in Python.

not_covering

Signature: *[const]* Region **not_covering** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which are not completely covering polygons from the other region

Returns: A new region containing the polygons which are not covering polygons from the other region

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This attribute is sometimes called 'enclosing' instead of 'covering', but this term is reserved for the respective DRC function.

This method has been introduced in version 0.27.

not_in

Signature: *[const]* Region **not_in** (const Region other)

Description: Returns all polygons which are not members of the other region

This method returns all polygons in self which can not be found in the other region with exactly the same geometry.

not_inside

Signature: *[const]* Region **not_inside** (const Region other)

Description: Returns the polygons of this region which are not completely inside polygons from the other region

Returns: A new region containing the polygons which are not inside polygons from the other region

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

not_interacting

(1) Signature: *[const]* Region **not_interacting** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which do not overlap or touch polygons from the other region

Returns: A new region containing the polygons not overlapping or touching polygons from the other region

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with (different) polygons of the other region to make the polygon not selected. A polygon is not selected by this method if the number of polygons interacting with a polygon of this region is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

The min_count and max_count arguments have been added in version 0.27.

(2) Signature: *[const]* Region **not_interacting** (const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which do not overlap or touch edges from the edge collection

Returns: A new region containing the polygons not overlapping or touching edges from the edge collection

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with edges of the edge collection to make the polygon not selected. A polygon is not selected by this method if the number of edges interacting with the polygon is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.25 The min_count and max_count arguments have been added in version 0.27.

(3) Signature: *[const]* Region **not_interacting** (const Texts other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which do not overlap or touch texts

Returns: A new region containing the polygons not overlapping or touching texts

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with texts of the text collection to make the polygon not selected. A polygon is not selected by this method if the number of texts interacting with the polygon is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27

not_members_of

Signature: *[const]* Region **not_members_of** (const Region other)

Description: Returns all polygons which are not members of the other region

This method returns all polygons in self which can not be found in the other region with exactly the same geometry.

not_outside

Signature: *[const]* Region **not_outside** (const Region other)

Description: Returns the polygons of this region which are not completely outside polygons from the other region

Returns:	A new region containing the polygons which are not outside polygons from the other region

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

not_overlapping

Signature: *[const]* Region **not_overlapping** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which do not overlap polygons from the other region

Returns:	A new region containing the polygons not overlapping polygons from the other region

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

The count options have been introduced in version 0.27.

not_with

Signature: Region **not_with** (const Region other, PropertyConstraint property_constraint = IgnoreProperties)

Description: Performs the boolean NOT between self and the other region in-place (modifying self)

Returns:	The region after modification (self)

This method will compute the boolean NOT (intersection) between two regions. The result is often but not necessarily always merged. It allows specification of a property constaint - e.g. only performing the boolean operation between shapes with the same user properties.

This variant has been introduced in version 0.28.4.

notch_check

Signature: *[const]* EdgePairs **notch_check** (unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs a space check between edges of the same polygon with options

d:	The minimum space for which the polygons are checked
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper limit of the projected length of one edge onto another
shielded:	Enables shielding

negative:	If true, edges not violation the condition will be output as pseudo-edge pairs
property_constraint:	Specifies whether to consider only shapes with a certain property relation
property_constraint:	Only IgnoreProperties and NoPropertyConstraint are allowed - in the last case, properties are copied from the original shapes to the output@param zero_distance_mode Specifies how to handle edges with zero distance

This version is similar to the simple version with one parameter. In addition, it allows to specify many more options.

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the space check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants.

"ignore_angle" specifies the angle limit of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one limit, pass nil to the respective value.

"shielded" controls whether shielding is applied. Shielding means that rule violations are not detected 'through' other features. Measurements are only made where the opposite edge is unobstructed. Shielding often is not optional as a rule violation in shielded case automatically comes with rule violations between the original and the shielding features. If not necessary, shielding can be disabled by setting this flag to false. In general, this will improve performance somewhat.

Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

The 'shielded' and 'negative' options have been introduced in version 0.27. 'property_constraint' has been added in version 0.28.4. 'zero_distance_mode' has been added in version 0.29.

or

Signature: *[const]* Region **or** (const Region other)

Description: Returns the boolean OR between self and the other region

Returns:	The resulting region

The boolean OR is implemented by merging the polygons of both regions. To simply join the regions without merging, the + operator is more efficient. The 'or' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'or_' in Python.

or_with

Signature: Region **or_with** (const Region other)

Description: Performs the boolean OR between self and the other region in-place (modifying self)

Returns:	The region after modification (self)

The boolean OR is implemented by merging the polygons of both regions. To simply join the regions without merging, the + operator is more efficient. Note that in Ruby, the '|=' operator actually does not exist, but is emulated by '|' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'or_with' instead.

The 'or_with' alias has been introduced in version 0.28.12.

outside

Signature: *[const]* Region **outside** (const Region other)

Description: Returns the polygons of this region which are completely outside polygons from the other region

Returns: A new region containing the polygons which are outside polygons from the other region

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

overlap_check

Signature: *[const]* EdgePairs **overlap_check** (const Region other, unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs an overlap check with options

d: The minimum overlap for which the polygons are checked

other: The other region against which to check

whole_edges: If true, deliver the whole edges

metrics: Specify the metrics type

ignore_angle: The angle above which no check is performed

min_projection: The lower threshold of the projected length of one edge onto another

max_projection: The upper limit of the projected length of one edge onto another

opposite_filter: Specifies a filter mode for errors happening on opposite sides of inputs shapes

rect_filter: Specifies an error filter for rectangular input shapes

negative: Negative output from the first input

property_constraint: Specifies whether to consider only shapes with a certain property relation

zero_distance_mode: Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants.

"ignore_angle" specifies the angle limit of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one limit, pass nil to the respective value.

"shielded" controls whether shielding is applied. Shielding means that rule violations are not detected 'through' other features. Measurements are only made where the opposite edge is unobstructed. Shielding often is not optional as a rule violation in shielded case automatically comes with rule violations between the original and the shielding features. If not necessary, shielding can be disabled by setting this flag to false. In general, this will improve performance somewhat.

"opposite_filter" specifies whether to require or reject errors happening on opposite sides of a figure. "rect_filter" allows suppressing specific error configurations on rectangular input figures.

If "negative" is true, only edges from the first input are output as pseudo edge-pairs where the overlap is larger or equal to the limit. This is a way to flag the parts of the first input where the overlap vs. the second input is bigger. Note that only the first input's edges are output. The output is still edge pairs,

but each edge pair contains one edge from the original input and the reverse version of the edge as the second edge.

Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

The 'shielded', 'negative', 'not_opposite' and 'rect_sides' options have been introduced in version 0.27. The interpretation of the 'negative' flag has been restriced to first-layout only output in 0.27.1. 'property_constraint' has been added in version 0.28.4. 'zero_distance_mode' has been added in version 0.29.

overlapping

Signature: *[const]* Region **overlapping** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which overlap polygons from the other region

Returns: A new region containing the polygons overlapping polygons from the other region

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

The count options have been introduced in version 0.27.

perimeter

(1) Signature: *[const]* unsigned long **perimeter**

Description: The total perimeter of the polygons

Merged semantics applies for this method (see merged_semantics= for a description of this concept) If merged semantics is not enabled, internal edges are counted as well.

(2) Signature: *[const]* unsigned long **perimeter** (const Box rect)

Description: The total perimeter of the polygons (restricted to a rectangle)

This version will compute the perimeter of the polygons, restricting the computation to the given rectangle. Edges along the border are handled in a special way: they are counted when they are oriented with their inside side toward the rectangle (in other words: outside edges must coincide with the rectangle's border in order to be counted).

Merged semantics applies for this method (see merged_semantics= for a description of this concept) If merged semantics is not enabled, internal edges are counted as well.

process

Signature: void **process** (const PolygonOperator ptr process)

Description: Applies a generic polygon processor in place (replacing the polygons from the Region)

See PolygonProcessor for a description of this feature.

This method has been introduced in version 0.29.

processed

(1) Signature: *[const]* Region **processed** (const PolygonOperator ptr processed)

Description: Applies a generic polygon processor and returns a processed copy

See PolygonProcessor for a description of this feature.

This method has been introduced in version 0.29.

(2) Signature: *[const]* EdgePairs **processed** (const PolygonToEdgePairOperator ptr processed)

Description: Applies a generic polygon-to-edge-pair processor and returns an edge pair collection with the results

See PolygonToEdgePairProcessor for a description of this feature.

This method has been introduced in version 0.29.

(3) Signature: *[const]* Edges **processed** (const PolygonToEdgeOperator ptr processed)

Description: Applies a generic polygon-to-edge processor and returns an edge collection with the results

See PolygonToEdgeProcessor for a description of this feature.

This method has been introduced in version 0.29.

pull_inside

Signature: *[const]* Region **pull_inside** (const Region other)

Description: Returns all polygons of "other" which are inside polygons of this region

Returns: The region after the polygons have been selected (from other)

The "pull_..." methods are similar to "select_..." but work the opposite way: they select shapes from the argument region rather than self. In a deep (hierarchical) context the output region will be hierarchically aligned with self, so the "pull_..." methods provide a way for re-hierarchization.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.26.1

pull_interacting

(1) Signature: *[const]* Region **pull_interacting** (const Region other)

Description: Returns all polygons of "other" which are interacting with (overlapping, touching) polygons of this region

Returns: The region after the polygons have been selected (from other)

See pull_inside for a description of the "pull_..." methods.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.26.1

(2) Signature: *[const]* Edges **pull_interacting** (const Edges other)

Description: Returns all edges of "other" which are interacting with polygons of this region

Returns: The edge collection after the edges have been selected (from other)

See pull_inside for a description of the "pull_..." methods.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.26.1

(3) Signature: *[const]* Texts **pull_interacting** (const Texts other)

Description: Returns all texts of "other" which are interacting with polygons of this region

Returns: The text collection after the texts have been selected (from other)

See pull_inside for a description of the "pull_..." methods.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27

pull_overlapping

Signature: *[const]* Region **pull_overlapping** (const Region other)

Description: Returns all polygons of "other" which are overlapping polygons of this region

Returns: The region after the polygons have been selected (from other)

See pull_inside for a description of the "pull_..." methods.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.26.1

rasterize

(1) Signature: *[const]* double[][] **rasterize** (const Point origin, const Vector pixel_size, unsigned int nx, unsigned int ny)

Description: A grayscale rasterizer delivering the area covered per pixel

origin:	The lower-left corner of the lowest-left pixel
pixel_size:	The dimension of each pixel (the x component gives the width, the y component the height)
nx:	The number of pixels in horizontal direction
ny:	The number of pixels in vertical direction

The method will create a grayscale, high-resolution density map of a rectangular region. The scan region is defined by the origin, the pixel size and the number of pixels in horizontal (nx) and vertical (ny) direction. The resulting array will contain the area covered by polygons from the region in square database units.

For non-overlapping polygons, the maximum density value is px*py. Overlapping polygons are counted multiple times, so the actual values may be larger. If you want overlaps removed, you have to merge the region before. Merge semantics does not apply for the 'rasterize' method.

The resulting area values are precise within the limits of double-precision floating point arithmetics.

A second version exists that allows specifying an active pixel size which is smaller than the pixel distance hence allowing pixels samples that do not cover the full area, but leave gaps between the pixels.

This method has been added in version 0.29.

(2) Signature: *[const]* double[][] **rasterize** (const Point origin, const Vector pixel_distance, const Vector pixel_size, unsigned int nx, unsigned int ny)

Description: A version of 'rasterize' that allows a pixel step distance which is larger than the pixel size

This version behaves like the first variant of 'rasterize', but the pixel distance (pixel-to-pixel step raster) can be specified separately from the pixel size. Currently, the pixel size must be equal or smaller than the pixel distance - i.e. the pixels must not overlap.

This method has been added in version 0.29.

rectangles

Signature: *[const]* Region **rectangles**

Description: Returns all polygons which are rectangles

This method returns all polygons in self which are rectangles.Merged semantics applies for this method (see merged_semantics= for a description of this concept)

rectilinear

Signature: *[const]* Region **rectilinear**

Description: Returns all polygons which are rectilinear

This method returns all polygons in self which are rectilinear.Merged semantics applies for this method (see merged_semantics= for a description of this concept)

remove_properties

Signature: void **remove_properties**

Description: Removes properties for the given container.

This will remove all properties on the given container.

This method has been introduced in version 0.28.4.

round_corners

Signature: void **round_corners** (double r_inner, double r_outer, unsigned int n)

Description: Corner rounding

r_inner:	Inner corner radius (in database units)

r_outer: Outer corner radius (in database units)

n: The number of points per circle

This method rounds the corners of the polygons in the region. Inner corners will be rounded with a radius of r_inner and outer corners with a radius of r_outer. The circles will be approximated by segments using n segments per full circle.

This method modifies the region. rounded_corners is a method that does the same but returns a new region without modifying self. Merged semantics applies for this method.

rounded_corners

Signature: *[const]* Region **rounded_corners** (double r_inner, double r_outer, unsigned int n)

Description: Corner rounding

r_inner: Inner corner radius (in database units)

r_outer: Outer corner radius (in database units)

n: The number of points per circle

See round_corners for a description of this method. This version returns a new region instead of modifying self (out-of-place).

scale_and_snap

Signature: void **scale_and_snap** (int gx, int mx, int dx, int gy, int my, int dy)

Description: Scales and snaps the region to the given grid

This method will first scale the region by a rational factor of mx/dx horizontally and my/dy vertically and then snap the region to the given grid - each x or y coordinate is brought on the gx or gy grid by rounding to the nearest value which is a multiple of gx or gy.

If gx or gy is 0, the result is brought on a grid of 1.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.26.1.

scaled_and_snapped

Signature: *[const]* Region **scaled_and_snapped** (int gx, int mx, int dx, int gy, int my, int dy)

Description: Returns the scaled and snapped region

This method will scale and snap the region to the given grid and return the scaled and snapped region (see scale_and_snap). The original region is not modified.

This method has been introduced in version 0.26.1.

select_covering

Signature: Region **select_covering** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the polygons of this region which are completely covering polygons from the other region

Returns: The region after the polygons have been selected (self)

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This attribute is sometimes called 'enclosing' instead of 'covering', but this term is reserved for the respective DRC function.

This method has been introduced in version 0.27.

select_inside

Signature: Region **select_inside** (const Region other)

Description: Selects the polygons of this region which are completely inside polygons from the other region

Returns: The region after the polygons have been selected (self)

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

select_interacting

(1) Signature: Region **select_interacting** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the polygons from this region which overlap or touch polygons from the other region

Returns: The region after the polygons have been selected (self)

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with (different) polygons of the other region to make the polygon selected. A polygon is selected by this method if the number of polygons interacting with a polygon of this region is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

The min_count and max_count arguments have been added in version 0.27.

(2) Signature: Region **select_interacting** (const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the polygons from this region which overlap or touch edges from the edge collection

Returns: The region after the polygons have been selected (self)

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with edges of the edge collection to make the polygon selected. A polygon is selected by this method if the number of edges interacting with the polygon is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.25 The min_count and max_count arguments have been added in version 0.27.

(3) Signature: Region **select_interacting** (const Texts other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the polygons of this region which overlap or touch texts

Returns: The region after the polygons have been selected (self)

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with texts of the text collection to make the polygon selected. A polygon is selected by this method if the number of texts interacting with the polygon is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27

select_not_covering

Signature: Region **select_not_covering** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the polygons of this region which are not completely covering polygons from the other region

Returns: The region after the polygons have been selected (self)

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This attribute is sometimes called 'enclosing' instead of 'covering', but this term is reserved for the respective DRC function.

This method has been introduced in version 0.27.

select_not_inside

Signature: Region **select_not_inside** (const Region other)

Description: Selects the polygons of this region which are not completely inside polygons from the other region

Returns: The region after the polygons have been selected (self)

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

select_not_interacting

(1) Signature: Region **select_not_interacting** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the polygons from this region which do not overlap or touch polygons from the other region

Returns: The region after the polygons have been selected (self)

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with (different) polygons of the other region to make the polygon not selected. A polygon is not selected by this method if the number of polygons interacting with a polygon of this region is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

The min_count and max_count arguments have been added in version 0.27.

(2) Signature: Region **select_not_interacting** (const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the polygons from this region which do not overlap or touch edges from the edge collection

Returns: The region after the polygons have been selected (self)

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with edges of the edge collection to make the polygon not selected. A polygon is not selected by this method if the number of edges interacting with the polygon is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.25 The min_count and max_count arguments have been added in version 0.27.

(3) Signature: Region **select_not_interacting** (const Texts other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the polygons of this region which do not overlap or touch texts

Returns: The region after the polygons have been selected (self)

'min_count' and 'max_count' impose a constraint on the number of times a polygon of this region has to interact with texts of the text collection to make the polygon not selected. A polygon is not selected by this method if the number of texts interacting with the polygon is between min_count and max_count (including max_count).

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27

select_not_outside

Signature: Region **select_not_outside** (const Region other)

Description: Selects the polygons of this region which are not completely outside polygons from the other region

Returns: The region after the polygons have been selected (self)

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

select_not_overlapping

Signature: Region **select_not_overlapping** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the polygons from this region which do not overlap polygons from the other region

Returns: The region after the polygons have been selected (self)

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

The count options have been introduced in version 0.27.

select_outside

Signature: Region **select_outside** (const Region other)

Description: Selects the polygons of this region which are completely outside polygons from the other region

Returns: The region after the polygons have been selected (self)

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

select_overlapping

Signature: Region **select_overlapping** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Selects the polygons from this region which overlap polygons from the other region

Returns: The region after the polygons have been selected (self)

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

The count options have been introduced in version 0.27.

separation_check

Signature: *[const]* EdgePairs **separation_check** (const Region other, unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs a separation check with options

d:	The minimum separation for which the polygons are checked
other:	The other region against which to check
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper limit of the projected length of one edge onto another
opposite_filter:	Specifies a filter mode for errors happening on opposite sides of inputs shapes
rect_filter:	Specifies an error filter for rectangular input shapes
negative:	Negative output from the first input
property_constraint:	Specifies whether to consider only shapes with a certain property relation
zero_distance_mode:	Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants.

"ignore_angle" specifies the angle limit of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one limit, pass nil to the respective value.

"shielded" controls whether shielding is applied. Shielding means that rule violations are not detected 'through' other features. Measurements are only made where the opposite edge is unobstructed. Shielding often is not optional as a rule violation in shielded case automatically comes with rule violations between the original and the shielding features. If not necessary, shielding can be disabled by setting this flag to false. In general, this will improve performance somewhat.

"opposite_filter" specifies whether to require or reject errors happening on opposite sides of a figure. "rect_filter" allows suppressing specific error configurations on rectangular input figures.

If "negative" is true, only edges from the first input are output as pseudo edge-pairs where the separation is larger or equal to the limit. This is a way to flag the parts of the first input where the distance to the second input is bigger. Note that only the first input's edges are output. The output is still edge pairs, but each edge pair contains one edge from the original input and the reverse version of the edge as the second edge.

Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

The 'shielded', 'negative', 'not_opposite' and 'rect_sides' options have been introduced in version 0.27. The interpretation of the 'negative' flag has been restriced to first-layout only output in 0.27.1. 'property_constraint' has been added in version 0.28.4. 'zero_distance_mode' has been added in version 0.29.

size

(1) Signature: Region **size** (int dx, int dy, unsigned int mode)

Description: Anisotropic sizing (biasing)

Returns: The region after the sizing has applied (self)

Shifts the contour outwards (dx,dy>0) or inwards (dx,dy<0). dx is the sizing in x-direction and dy is the sizing in y-direction. The sign of dx and dy should be identical.

This method applies a sizing to the region. Before the sizing is done, the region is merged if this is not the case already.

The mode defines at which bending angle cutoff occurs (0:>0, 1:>45, 2:>90, 3:>135, 4:>approx. 168, other:>approx. 179)

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

The result is a set of polygons which may be overlapping, but are not self- intersecting. Polygons may overlap afterwards because they grew big enough to overlap their neighbors. In that case, merge can be used to detect this overlaps by setting the "min_wc" parameter to value 1:

```
r = RBA::Region::new
r.insert(RBA::Box::new(0, 0, 50, 50))
r.insert(RBA::Box::new(100, 0, 150, 50))
r.size(50, 2)
r.merge(false, 1)
# r now is (50,-50;50,100;100,100;100,-50)
```

(2) Signature: Region **size** (const Vector dv, unsigned int mode = 2)

Description: Anisotropic sizing (biasing)

Returns: The region after the sizing has applied (self)

This method is equivalent to "size(dv.x, dv.y, mode)".

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This variant has been introduced in version 0.28.

(3) Signature: Region **size** (int d, unsigned int mode = 2)

Description: Isotropic sizing (biasing)

Returns: The region after the sizing has applied (self)

This method is equivalent to "size(d, d, mode)".

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

(4) Signature: *[const]* unsigned long **size**

Description: Returns the (flat) number of polygons in the region

Use of this method is deprecated. Use count instead

This returns the number of raw polygons (not merged polygons if merged semantics is enabled). The count is computed 'as if flat', i.e. polygons inside a cell are multiplied by the number of times a cell is instantiated.

The 'count' alias has been provided in version 0.26 to avoid ambiguity with the 'size' method which applies a geometrical bias.

Python specific notes:
This method is also available as 'len(object)'.

sized

(1) Signature: *[const]* Region **sized** (int dx, int dy, unsigned int mode)

Description: Returns the anisotropically sized region

Returns: The sized region

This method returns the sized region (see size), but does not modify self.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

(2) Signature: *[const]* Region **sized** (const Vector dv, unsigned int mode = 2)

Description: Returns the (an)isotropically sized region

Returns: The sized region

This method is equivalent to "sized(dv.x, dv.y, mode)". This method returns the sized region (see size), but does not modify self.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This variant has been introduced in version 0.28.

(3) Signature: *[const]* Region **sized** (int d, unsigned int mode = 2)

Description: Returns the isotropically sized region

Returns: The sized region

This method is equivalent to "sized(d, d, mode)". This method returns the sized region (see size), but does not modify self.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

smooth

Signature: void **smooth** (int d, bool keep_hv = false)

Description: Smoothing

d:	The smoothing tolerance (in database units)
keep_hv:	If true, horizontal and vertical edges are maintained

This method will simplify the merged polygons of the region by removing vertexes if the resulting polygon stays equivalent with the original polygon. Equivalence is measured in terms of a deviation which is guaranteed to not become larger than d. This method modifies the region. smoothed is a method that does the same but returns a new region without modifying self. Merged semantics applies for this method.

smoothed

Signature: *[const]* Region **smoothed** (int d, bool keep_hv = false)

Description: Smoothing

d:	The smoothing tolerance (in database units)
keep_hv:	If true, horizontal and vertical edges are maintained

See smooth for a description of this method. This version returns a new region instead of modifying self (out-of-place). It has been introduced in version 0.25.

snap

Signature: void **snap** (int gx, int gy)

Description: Snaps the region to the given grid

This method will snap the region to the given grid - each x or y coordinate is brought on the gx or gy grid by rounding to the nearest value which is a multiple of gx or gy.

If gx or gy is 0, no snapping happens in that direction.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

snapped

Signature: *[const]* Region **snapped** (int gx, int gy)

Description: Returns the snapped region

This method will snap the region to the given grid and return the snapped region (see snap). The original region is not modified.

space_check

Signature: *[const]* EdgePairs **space_check** (unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, Region::OppositeFilter opposite_filter = NoOppositeFilter, Region::RectFilter rect_filter = NoRectFilter, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs a space check with options

d:	The minimum space for which the polygons are checked
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper limit of the projected length of one edge onto another

opposite_filter:	Specifies a filter mode for errors happening on opposite sides of inputs shapes
rect_filter:	Specifies an error filter for rectangular input shapes
negative:	If true, edges not violation the condition will be output as pseudo-edge pairs
property_constraint:	Specifies whether to consider only shapes with a certain property relation
zero_distance_mode:	Specifies how to handle edges with zero distance

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants.

"ignore_angle" specifies the angle limit of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one limit, pass nil to the respective value.

"shielded" controls whether shielding is applied. Shielding means that rule violations are not detected 'through' other features. Measurements are only made where the opposite edge is unobstructed. Shielding often is not optional as a rule violation in shielded case automatically comes with rule violations between the original and the shielding features. If not necessary, shielding can be disabled by setting this flag to false. In general, this will improve performance somewhat.

"opposite_filter" specifies whether to require or reject errors happening on opposite sides of a figure. "rect_filter" allows suppressing specific error configurations on rectangular input figures.

Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

The 'shielded', 'negative', 'not_opposite' and 'rect_sides' options have been introduced in version 0.27. 'property_constraint' has been added in version 0.28.4. 'zero_distance_mode' has been added in version 0.29.

split_covering

Signature: *[const]* Region[] **split_covering** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which are completely covering polygons from the other region and the ones which are not at the same time

Returns:	Two new regions: the first containing the result of covering, the second the result of not_covering

This method is equivalent to calling covering and not_covering, but is faster when both results are required. Merged semantics applies for this method (see merged_semantics= for a description of this concept).

This method has been introduced in version 0.27.

split_inside

Signature: *[const]* Region[] **split_inside** (const Region other)

Description: Returns the polygons of this region which are completely inside polygons from the other region and the ones which are not at the same time

Returns:	Two new regions: the first containing the result of inside, the second the result of not_inside

This method is equivalent to calling inside and not_inside, but is faster when both results are required. Merged semantics applies for this method (see merged_semantics= for a description of this concept).

This method has been introduced in version 0.27.

split_interacting

(1) Signature: *[const]* Region[] **split_interacting** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which are interacting with polygons from the other region and the ones which are not at the same time

Returns: Two new regions: the first containing the result of interacting, the second the result of not_interacting

This method is equivalent to calling interacting and not_interacting, but is faster when both results are required. Merged semantics applies for this method (see merged_semantics= for a description of this concept).

This method has been introduced in version 0.27.

(2) Signature: *[const]* Region[] **split_interacting** (const Edges other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which are interacting with edges from the other edge collection and the ones which are not at the same time

Returns: Two new regions: the first containing the result of interacting, the second the result of not_interacting

This method is equivalent to calling interacting and not_interacting, but is faster when both results are required. Merged semantics applies for this method (see merged_semantics= for a description of this concept).

This method has been introduced in version 0.27.

(3) Signature: *[const]* Region[] **split_interacting** (const Texts other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which are interacting with texts from the other text collection and the ones which are not at the same time

Returns: Two new regions: the first containing the result of interacting, the second the result of not_interacting

This method is equivalent to calling interacting and not_interacting, but is faster when both results are required. Merged semantics applies for this method (see merged_semantics= for a description of this concept).

This method has been introduced in version 0.27.

split_outside

Signature: *[const]* Region[] **split_outside** (const Region other)

Description: Returns the polygons of this region which are completely outside polygons from the other region and the ones which are not at the same time

Returns: Two new regions: the first containing the result of outside, the second the result of not_outside

This method is equivalent to calling outside and not_outside, but is faster when both results are required. Merged semantics applies for this method (see merged_semantics= for a description of this concept).

This method has been introduced in version 0.27.

split_overlapping

Signature: *[const]* Region[] **split_overlapping** (const Region other, unsigned long min_count = 1, unsigned long max_count = unlimited)

Description: Returns the polygons of this region which are overlapping with polygons from the other region and the ones which are not at the same time

Returns: Two new regions: the first containing the result of overlapping, the second the result of not_overlapping

This method is equivalent to calling overlapping and not_overlapping, but is faster when both results are required. Merged semantics applies for this method (see merged_semantics= for a description of this concept).

This method has been introduced in version 0.27.

squares

Signature: *[const]* Region **squares**

Description: Returns all polygons which are squares

This method returns all polygons in self which are squares.Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27.

strange_polygon_check

Signature: *[const]* Region **strange_polygon_check**

Description: Returns a region containing those parts of polygons which are "strange"

Strange parts of polygons are self-overlapping parts or non-orientable parts (i.e. in the "8" configuration).

Merged semantics does not apply for this method (see merged_semantics= for a description of this concept)

strict_handling=

Signature: void **strict_handling=** (bool f)

Description: Enables or disables strict handling

Strict handling means to leave away some optimizations. Specifically the output of boolean operations will be merged even if one input is empty. Without strict handling, the operation will be optimized and output won't be merged.

Strict handling is disabled by default and optimization is in place.

This method has been introduced in version 0.23.2.

Python specific notes:
The object exposes a writable attribute 'strict_handling'. This is the setter.

strict_handling?

Signature: *[const]* bool **strict_handling?**

Description: Gets a flag indicating whether merged semantics is enabled

See strict_handling= for a description of this attribute.

This method has been introduced in version 0.23.2.

Python specific notes:
The object exposes a readable attribute 'strict_handling'. This is the getter.

swap

Signature: void **swap** (Region other)

Description: Swap the contents of this region with the contents of another region

This method is useful to avoid excessive memory allocation in some cases. For managed memory languages such as Ruby, those cases will be rare.

to_s

(1) Signature: *[const]* string **to_s**

Description: Converts the region to a string

The length of the output is limited to 20 polygons to avoid giant strings on large regions. For full output use "to_s" with a maximum count parameter.

Python specific notes:
This method is also available as 'str(object)'.

(2) Signature: *[const]* string **to_s** (unsigned long max_count)

Description: Converts the region to a string

This version allows specification of the maximum number of polygons contained in the string.

transform

(1) Signature: Region **transform** (const Trans t)

Description: Transform the region (modifies self)

t: The transformation to apply.

Returns: The transformed region.

Transforms the region with the given transformation. This version modifies the region and returns a reference to self.

(2) Signature: Region **transform** (const ICplxTrans t)

Description: Transform the region with a complex transformation (modifies self)

t: The transformation to apply.

Returns: The transformed region.

Transforms the region with the given transformation. This version modifies the region and returns a reference to self.

(3) Signature: Region **transform** (const IMatrix2d t)

Description: Transform the region (modifies self)

t: The transformation to apply.

Returns: The transformed region.

Transforms the region with the given 2d matrix transformation. This version modifies the region and returns a reference to self.

This variant was introduced in version 0.27.

(4) Signature: Region **transform** (const IMatrix3d t)

Description: Transform the region (modifies self)

t: The transformation to apply.

Returns: The transformed region.

Transforms the region with the given 3d matrix transformation. This version modifies the region and returns a reference to self.

This variant was introduced in version 0.27.

transform_icplx

Signature: Region **transform_icplx** (const ICplxTrans t)

Description: Transform the region with a complex transformation (modifies self)

t: The transformation to apply.

Returns: The transformed region.

Use of this method is deprecated. Use transform instead

Transforms the region with the given transformation. This version modifies the region and returns a reference to self.

transformed

(1) Signature: *[const]* Region **transformed** (const Trans t)

Description: Transforms the region

t: The transformation to apply.

Returns: The transformed region.

Transforms the region with the given transformation. Does not modify the region but returns the transformed region.

(2) Signature: *[const]* Region **transformed** (const ICplxTrans t)

Description: Transforms the region with a complex transformation

t: The transformation to apply.

Returns: The transformed region.

Transforms the region with the given complex transformation. Does not modify the region but returns the transformed region.

(3) Signature: *[const]* Region **transformed** (const IMatrix2d t)

Description: Transforms the region

t: The transformation to apply.

Returns: The transformed region.

Transforms the region with the given 2d matrix transformation. Does not modify the region but returns the transformed region.

This variant was introduced in version 0.27.

(4) Signature: *[const]* Region **transformed** (const IMatrix3d t)

Description: Transforms the region

t: The transformation to apply.

Returns: The transformed region.

Transforms the region with the given 3d matrix transformation. Does not modify the region but returns the transformed region.

This variant was introduced in version 0.27.

transformed_icplx

Signature: *[const]* Region **transformed_icplx** (const ICplxTrans t)

Description: Transforms the region with a complex transformation

t: The transformation to apply.

Returns: The transformed region.

Use of this method is deprecated. Use transformed instead

Transforms the region with the given complex transformation. Does not modify the region but returns the transformed region.

width_check

Signature: *[const]* EdgePairs **width_check** (unsigned int d, bool whole_edges = false, Metrics metrics = Euclidian, variant ignore_angle = default, variant min_projection = 0, variant max_projection = max, bool shielded = true, bool negative = false, PropertyConstraint property_constraint = IgnoreProperties, ZeroDistanceMode zero_distance_mode = IncludeZeroDistanceWhenTouching)

Description: Performs a width check with options

d:	The minimum width for which the polygons are checked
whole_edges:	If true, deliver the whole edges
metrics:	Specify the metrics type
ignore_angle:	The angle above which no check is performed
min_projection:	The lower threshold of the projected length of one edge onto another
max_projection:	The upper limit of the projected length of one edge onto another
shielded:	Enables shielding
negative:	If true, edges not violation the condition will be output as pseudo-edge pairs
property_constraint:	Only IgnoreProperties and NoPropertyConstraint are allowed - in the last case, properties are copied from the original shapes to the output. @param zero_distance_mode Specifies how to handle edges with zero distance

Other than 'width' allow more options here.

This version is similar to the simple version with one parameter. In addition, it allows to specify many more options.

If "whole_edges" is true, the resulting EdgePairs collection will receive the whole edges which contribute in the width check.

"metrics" can be one of the constants Euclidian, Square or Projection. See there for a description of these constants.

"ignore_angle" specifies the angle limit of two edges. If two edges form an angle equal or above the given value, they will not contribute in the check. Setting this value to 90 (the default) will exclude edges with an angle of 90 degree or more from the check. Use nil for this value to select the default.

"min_projection" and "max_projection" allow selecting edges by their projected value upon each other. It is sufficient if the projection of one edge on the other matches the specified condition. The projected length must be larger or equal to "min_projection" and less than "max_projection". If you don't want to specify one limit, pass nil to the respective value.

"shielded" controls whether shielding is applied. Shielding means that rule violations are not detected 'through' other features. Measurements are only made where the opposite edge is unobstructed. Shielding often is not optional as a rule violation in shielded case automatically comes with rule violations between the original and the shielding features. If not necessary, shielding can be disabled by setting this flag to false. In general, this will improve performance somewhat.

Merged semantics applies for the input of this method (see merged_semantics= for a description of this concept)

The 'shielded' and 'negative' options have been introduced in version 0.27. 'property_constraint' has been added in version 0.28.4. 'zero_distance_mode' has been added in version 0.28.16.

with_angle

(1) Signature: *[const]* EdgePairs **with_angle** (double angle, bool inverse)

Description: Returns markers on every corner with the given angle (or not with the given angle)

If the inverse flag is false, this method returns an error marker (an EdgePair object) for every corner whose connected edges form an angle with the given value (in degree). If the inverse flag is true, the method returns markers for every corner whose angle is not the given value.

The edge pair objects returned will contain both edges forming the angle.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

(2) Signature: *[const]* EdgePairs **with_angle** (double amin, double amax, bool inverse)

Description: Returns markers on every corner with an angle of more than amin and less than amax (or the opposite)

If the inverse flag is false, this method returns an error marker (an EdgePair object) for every corner whose connected edges form an angle whose value is more or equal to amin (in degree) or less (but not equal to) amax. If the inverse flag is true, the method returns markers for every corner whose angle is not matching that criterion.

The edge pair objects returned will contain both edges forming the angle.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

with_area

(1) Signature: *[const]* Region **with_area** (long area, bool inverse)

Description: Filter the polygons by area

Filters the polygons of the region by area. If "inverse" is false, only polygons which have the given area are returned. If "inverse" is true, polygons not having the given area are returned.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

(2) Signature: *[const]* Region **with_area** (variant min_area, variant max_area, bool inverse)

Description: Filter the polygons by area

Filters the polygons of the region by area. If "inverse" is false, only polygons which have an area larger or equal to "min_area" and less than "max_area" are returned. If "inverse" is true, polygons having an area less than "min_area" or larger or equal than "max_area" are returned.

If you don't want to specify a lower or upper limit, pass nil to that parameter.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

with_area_ratio

(1) Signature: *[const]* Region **with_area_ratio** (double ratio, bool inverse)

Description: Filters the polygons by the bounding box area to polygon area ratio

The area ratio is defined by the ratio of bounding box area to polygon area. It's a measure how much the bounding box is approximating the polygon. 'Thin polygons' have a large area ratio, boxes has an area ratio of 1. The area ratio is always larger or equal to 1. With 'inverse' set to false, this version filters polygons which have an area ratio equal to the given value. With 'inverse' set to true, all other polygons will be returned.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27.

(2) Signature: *[const]* Region **with_area_ratio** (variant min_ratio, variant max_ratio, bool inverse, bool min_included = true, bool max_included = true)

Description: Filters the polygons by the aspect ratio of their bounding boxes

The area ratio is defined by the ratio of bounding box area to polygon area. It's a measure how much the bounding box is approximating the polygon. 'Thin polygons' have a large area ratio, boxes has an area ratio of 1. The area ratio is always larger or equal to 1. With 'inverse' set to false, this version filters polygons which have an area ratio between 'min_ratio' and 'max_ratio'. With 'min_included' set to true, the 'min_ratio' value is included in the range, otherwise it's excluded. Same for 'max_included' and 'max_ratio'. With 'inverse' set to true, all other polygons will be returned.

If you don't want to specify a lower or upper limit, pass nil to that parameter.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27.

with_bbox_aspect_ratio

(1) Signature: *[const]* Region **with_bbox_aspect_ratio** (double ratio, bool inverse)

Description: Filters the polygons by the aspect ratio of their bounding boxes

Filters the polygons of the region by the aspect ratio of their bounding boxes. The aspect ratio is the ratio of larger to smaller dimension of the bounding box. A square has an aspect ratio of 1.

With 'inverse' set to false, this version filters polygons which have a bounding box aspect ratio equal to the given value. With 'inverse' set to true, all other polygons will be returned.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27.

(2) Signature: *[const]* Region **with_bbox_aspect_ratio** (variant min_ratio, variant max_ratio, bool inverse, bool min_included = true, bool max_included = true)

Description: Filters the polygons by the aspect ratio of their bounding boxes

Filters the polygons of the region by the aspect ratio of their bounding boxes. The aspect ratio is the ratio of larger to smaller dimension of the bounding box. A square has an aspect ratio of 1.

With 'inverse' set to false, this version filters polygons which have a bounding box aspect ratio between 'min_ratio' and 'max_ratio'. With 'min_included' set to true, the 'min_ratio' value is included in the range, otherwise it's excluded. Same for 'max_included' and 'max_ratio'. With 'inverse' set to true, all other polygons will be returned.

If you don't want to specify a lower or upper limit, pass nil to that parameter.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27.

with_bbox_height

(1) Signature: *[const]* Region **with_bbox_height** (unsigned int height, bool inverse)

Description: Filter the polygons by bounding box height

Filters the polygons of the region by the height of their bounding box. If "inverse" is false, only polygons whose bounding box has the given height are returned. If "inverse" is true, polygons whose bounding box does not have the given height are returned.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

(2) Signature: *[const]* Region **with_bbox_height** (variant min_height, variant max_height, bool inverse)

Description: Filter the polygons by bounding box height

Filters the polygons of the region by the height of their bounding box. If "inverse" is false, only polygons whose bounding box has a height larger or equal to "min_height" and less than "max_height" are returned. If "inverse" is true, all polygons not matching this criterion are returned. If you don't want to specify a lower or upper limit, pass nil to that parameter.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

with_bbox_max

(1) Signature: *[const]* Region **with_bbox_max** (unsigned int dim, bool inverse)

Description: Filter the polygons by bounding box width or height, whichever is larger

Filters the polygons of the region by the maximum dimension of their bounding box. If "inverse" is false, only polygons whose bounding box's larger dimension is equal to the given value are returned. If "inverse" is true, all polygons not matching this criterion are returned. Merged semantics applies for this method (see merged_semantics= for a description of this concept)

(2) Signature: *[const]* Region **with_bbox_max** (variant min_dim, variant max_dim, bool inverse)

Description: Filter the polygons by bounding box width or height, whichever is larger

Filters the polygons of the region by the minimum dimension of their bounding box. If "inverse" is false, only polygons whose bounding box's larger dimension is larger or equal to "min_dim" and less than "max_dim" are returned. If "inverse" is true, all polygons not matching this criterion are returned. If you don't want to specify a lower or upper limit, pass nil to that parameter.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

with_bbox_min

(1) Signature: *[const]* Region **with_bbox_min** (unsigned int dim, bool inverse)

Description: Filter the polygons by bounding box width or height, whichever is smaller

Filters the polygons inside the region by the minimum dimension of their bounding box. If "inverse" is false, only polygons whose bounding box's smaller dimension is equal to the given value are returned. If "inverse" is true, all polygons not matching this criterion are returned. Merged semantics applies for this method (see merged_semantics= for a description of this concept)

(2) Signature: *[const]* Region **with_bbox_min** (variant min_dim, variant max_dim, bool inverse)

Description: Filter the polygons by bounding box width or height, whichever is smaller

Filters the polygons of the region by the minimum dimension of their bounding box. If "inverse" is false, only polygons whose bounding box's smaller dimension is larger or equal to "min_dim" and less than "max_dim" are returned. If "inverse" is true, all polygons not matching this criterion are returned. If you don't want to specify a lower or upper limit, pass nil to that parameter.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

with_bbox_width

(1) Signature: *[const]* Region **with_bbox_width** (unsigned int width, bool inverse)

Description: Filter the polygons by bounding box width

Filters the polygons of the region by the width of their bounding box. If "inverse" is false, only polygons whose bounding box has the given width are returned. If "inverse" is true, polygons whose bounding box does not have the given width are returned.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

(2) Signature: *[const]* Region **with_bbox_width** (variant min_width, variant max_width, bool inverse)

Description: Filter the polygons by bounding box width

Filters the polygons of the region by the width of their bounding box. If "inverse" is false, only polygons whose bounding box has a width larger or equal to "min_width" and less than "max_width" are returned. If "inverse" is true, all polygons not matching this criterion are returned. If you don't want to specify a lower or upper limit, pass nil to that parameter.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

with_holes

(1) Signature: *[const]* Region **with_holes** (unsigned long nholes, bool inverse)

Description: Filters the polygons by their number of holes

Filters the polygons of the region by number of holes. If "inverse" is false, only polygons which have the given number of holes are returned. If "inverse" is true, polygons not having the given of holes are returned.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27.

(2) Signature: *[const]* Region **with_holes** (variant min_bholes, variant max_nholes, bool inverse)

Description: Filter the polygons by their number of holes

Filters the polygons of the region by number of holes. If "inverse" is false, only polygons which have a hole count larger or equal to "min_nholes" and less than "max_nholes" are returned. If "inverse" is true, polygons having a hole count less than "min_nholes" or larger or equal than "max_nholes" are returned.

If you don't want to specify a lower or upper limit, pass nil to that parameter.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27.

with_perimeter

(1) Signature: *[const]* Region **with_perimeter** (unsigned long perimeter, bool inverse)

Description: Filter the polygons by perimeter

Filters the polygons of the region by perimeter. If "inverse" is false, only polygons which have the given perimeter are returned. If "inverse" is true, polygons not having the given perimeter are returned.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

(2) Signature: *[const]* Region **with_perimeter** (variant min_perimeter, variant max_perimeter, bool inverse)

Description: Filter the polygons by perimeter

Filters the polygons of the region by perimeter. If "inverse" is false, only polygons which have a perimeter larger or equal to "min_perimeter" and less than "max_perimeter" are returned. If "inverse" is true, polygons having a perimeter less than "min_perimeter" or larger or equal than "max_perimeter" are returned.

If you don't want to specify a lower or upper limit, pass nil to that parameter.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

with_relative_height

(1) Signature: *[const]* Region **with_relative_height** (double ratio, bool inverse)

Description: Filters the polygons by the ratio of height to width

This method filters the polygons of the region by the ratio of height vs. width of their bounding boxes. 'Tall' polygons have a large value while 'flat' polygons have a small value. A square has a relative height of 1.

An alternative method is 'with_area_ratio' which can be more efficient because it's isotropic.

With 'inverse' set to false, this version filters polygons which have a relative height equal to the given value. With 'inverse' set to true, all other polygons will be returned.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27.

(2) Signature: *[const]* Region **with_relative_height** (variant min_ratio, variant max_ratio, bool inverse, bool min_included = true, bool max_included = true)

Description: Filters the polygons by the bounding box height to width ratio

This method filters the polygons of the region by the ratio of height vs. width of their bounding boxes. 'Tall' polygons have a large value while 'flat' polygons have a small value. A square has a relative height of 1.

An alternative method is 'with_area_ratio' which can be more efficient because it's isotropic.

With 'inverse' set to false, this version filters polygons which have a relative height between 'min_ratio' and 'max_ratio'. With 'min_included' set to true, the 'min_ratio' value is included in the range, otherwise it's excluded. Same for 'max_included' and 'max_ratio'. With 'inverse' set to true, all other polygons will be returned.

If you don't want to specify a lower or upper limit, pass nil to that parameter.

Merged semantics applies for this method (see merged_semantics= for a description of this concept)

This method has been introduced in version 0.27.

write

Signature: *[const]* void **write** (string filename)

Description: Writes the region to a file

This method is provided for debugging purposes. It writes the object to a flat layer 0/0 in a single top cell.

This method has been introduced in version 0.29.

xor

Signature: *[const]* Region **xor** (const Region other)

Description: Returns the boolean XOR between self and the other region

Returns: The result of the boolean XOR operation

This method will compute the boolean XOR (intersection) between two regions. The result is often but not necessarily always merged.

The 'xor' alias has been introduced in version 0.28.12.

xor_with

Signature: Region **xor_with** (const Region other)

Description: Performs the boolean XOR between self and the other region in-place (modifying self)

Returns: The region after modification (self)

This method will compute the boolean XOR (intersection) between two regions. The result is often but not necessarily always merged.

Note that in Ruby, the '^=' operator actually does not exist, but is emulated by '^' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'xor_with' instead.

The 'xor_with' alias has been introduced in version 0.28.12.

|

Signature: *[const]* Region **|** (const Region other)

Description: Returns the boolean OR between self and the other region

Returns: The resulting region

The boolean OR is implemented by merging the polygons of both regions. To simply join the regions without merging, the + operator is more efficient. The 'or' alias has been introduced in version 0.28.12.

Python specific notes:
This attribute is available as 'or_' in Python.

|=

Signature: Region **|=** (const Region other)

Description: Performs the boolean OR between self and the other region in-place (modifying self)

Returns: The region after modification (self)

The boolean OR is implemented by merging the polygons of both regions. To simply join the regions without merging, the + operator is more efficient. Note that in Ruby, the '|=' operator actually does not exist, but is emulated by '|' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'or_with' instead.

The 'or_with' alias has been introduced in version 0.28.12.

4.88. API reference - Class Region::RectFilter

Notation used in Ruby API documentation

Module: db

Description: This class represents the error filter mode on rectangles for \Region#separation and related checks.

This class is equivalent to the class Region::RectFilter

This enum has been introduced in version 0.27.

Public constructors

new Region::RectFilter ptr	**new**	(int i)	Creates an enum from an integer value
new Region::RectFilter ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const Region::RectFilter other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const Region::RectFilter other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const Region::RectFilter other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	Region::RectFilter	**FourSidesAllowed**	Allow errors when on all sides
[static,const]	Region::RectFilter	**NoRectFilter**	Specifies no filtering
[static,const]	Region::RectFilter	**OneSideAllowed**	Allow errors on one side
[static,const]	Region::RectFilter	**ThreeSidesAllowed**	Allow errors when on three sides

[static,const]	Region::RectFilter	**TwoConnectedSidesAllow**	Allow errors on two sides ("L" configuration)
[static,const]	Region::RectFilter	**TwoOppositeSidesAllowed**	Allow errors on two opposite sides
[static,const]	Region::RectFilter	**TwoSidesAllowed**	Allow errors on two sides (not specified which)

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const Region::RectFilter other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const Region::RectFilter other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const Region::RectFilter other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
FourSidesAllowed	**Signature**: *[static,const]* Region::RectFilter **FourSidesAllowed** **Description**: Allow errors when on all sides **Python specific notes:** The object exposes a readable attribute 'FourSidesAllowed'. This is the getter.
NoRectFilter	**Signature**: *[static,const]* Region::RectFilter **NoRectFilter** **Description**: Specifies no filtering **Python specific notes:** The object exposes a readable attribute 'NoRectFilter'. This is the getter.
OneSideAllowed	**Signature**: *[static,const]* Region::RectFilter **OneSideAllowed** **Description**: Allow errors on one side **Python specific notes:** The object exposes a readable attribute 'OneSideAllowed'. This is the getter.
ThreeSidesAllowed	**Signature**: *[static,const]* Region::RectFilter **ThreeSidesAllowed** **Description**: Allow errors when on three sides **Python specific notes:**

	The object exposes a readable attribute 'ThreeSidesAllowed'. This is the getter.
TwoConnectedSidesAllowed	**Signature**: *[static,const]* Region::RectFilter **TwoConnectedSidesAllowed** **Description**: Allow errors on two sides ("L" configuration) **Python specific notes:** The object exposes a readable attribute 'TwoConnectedSidesAllowed'. This is the getter.
TwoOppositeSidesAllowed	**Signature**: *[static,const]* Region::RectFilter **TwoOppositeSidesAllowed** **Description**: Allow errors on two opposite sides **Python specific notes:** The object exposes a readable attribute 'TwoOppositeSidesAllowed'. This is the getter.
TwoSidesAllowed	**Signature**: *[static,const]* Region::RectFilter **TwoSidesAllowed** **Description**: Allow errors on two sides (not specified which) **Python specific notes:** The object exposes a readable attribute 'TwoSidesAllowed'. This is the getter.
hash	**Signature**: *[const]* int **hash** **Description**: Gets the hash value from the enum **Python specific notes:** This method is also available as 'hash(object)'.
inspect	**Signature**: *[const]* string **inspect** **Description**: Converts an enum to a visual string **Python specific notes:** This method is also available as 'repr(object)'.
new	**(1) Signature**: *[static]* new Region::RectFilter ptr **new** (int i) **Description**: Creates an enum from an integer value **Python specific notes:** This method is the default initializer of the object. **(2) Signature**: *[static]* new Region::RectFilter ptr **new** (string s) **Description**: Creates an enum from a string value **Python specific notes:** This method is the default initializer of the object.
to_i	**Signature**: *[const]* int **to_i** **Description**: Gets the integer value from the enum **Python specific notes:** This method is also available as 'int(object)'.
to_s	**Signature**: *[const]* string **to_s** **Description**: Gets the symbolic string from an enum **Python specific notes:** This method is also available as 'str(object)'.

4.89. API reference - Class Region::OppositeFilter

Notation used in Ruby API documentation

Module: db

Description: This class represents the opposite error filter mode for \Region#separation and related checks.

This class is equivalent to the class Region::OppositeFilter

This enum has been introduced in version 0.27.

Public constructors

new Region::OppositeFilter ptr	**new**	(int i)	Creates an enum from an integer value
new Region::OppositeFilter ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const Region::OppositeFilter other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const Region::OppositeFilter other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const Region::OppositeFilter other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	Region::OppositeFilter	**NoOppositeFilter**	No opposite filtering
[static,const]	Region::OppositeFilter	**NotOpposite**	Only errors NOT appearing on opposite sides of a figure will be reported
[static,const]	Region::OppositeFilter	**OnlyOpposite**	Only errors appearing on opposite sides of a figure will be reported

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const Region::OppositeFilter other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const Region::OppositeFilter other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const Region::OppositeFilter other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
NoOppositeFilter	**Signature**: *[static,const]* Region::OppositeFilter **NoOppositeFilter** **Description**: No opposite filtering **Python specific notes:** The object exposes a readable attribute 'NoOppositeFilter'. This is the getter.
NotOpposite	**Signature**: *[static,const]* Region::OppositeFilter **NotOpposite** **Description**: Only errors NOT appearing on opposite sides of a figure will be reported **Python specific notes:** The object exposes a readable attribute 'NotOpposite'. This is the getter.
OnlyOpposite	**Signature**: *[static,const]* Region::OppositeFilter **OnlyOpposite** **Description**: Only errors appearing on opposite sides of a figure will be reported **Python specific notes:** The object exposes a readable attribute 'OnlyOpposite'. This is the getter.
hash	**Signature**: *[const]* int **hash** **Description**: Gets the hash value from the enum **Python specific notes:** This method is also available as 'hash(object)'.
inspect	**Signature**: *[const]* string **inspect** **Description**: Converts an enum to a visual string **Python specific notes:** This method is also available as 'repr(object)'.

new

(1) Signature: *[static]* new Region::OppositeFilter ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Region::OppositeFilter ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.90. API reference - Class Metrics

Notation used in Ruby API documentation

Module: db

Description: This class represents the metrics type for \Region#width and related checks.

This enum has been introduced in version 0.27.

Public constructors

new Metrics ptr	**new**	(int i)		Creates an enum from an integer value
new Metrics ptr	**new**	(string s)		Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const Metrics other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const Metrics other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const Metrics other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Metrics other)	Assigns another object to self
[const]	new Metrics ptr	**dup**		Creates a copy of self
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string

[const]	int	**to_i**	Gets the integer value from the enum
[const]	string	**to_s**	Gets the symbolic string from an enum

Public static methods and constants

[static,const]	Metrics	**Euclidian**	Specifies Euclidian metrics for the check functions
[static,const]	Metrics	**Projection**	Specifies projected distance metrics for the check functions
[static,const]	Metrics	**Square**	Specifies square metrics for the check functions

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

(1) Signature: *[const]* bool **!=** (const Metrics other)
Description: Compares two enums for inequality

(2) Signature: *[const]* bool **!=** (int other)
Description: Compares an enum with an integer for inequality

<

(1) Signature: *[const]* bool **<** (const Metrics other)
Description: Returns true if the first enum is less (in the enum symbol order) than the second

(2) Signature: *[const]* bool **<** (int other)
Description: Returns true if the enum is less (in the enum symbol order) than the integer value

==

(1) Signature: *[const]* bool **==** (const Metrics other)
Description: Compares two enums

(2) Signature: *[const]* bool **==** (int other)
Description: Compares an enum with an integer value

Euclidian

Signature: *[static,const]* Metrics **Euclidian**
Description: Specifies Euclidian metrics for the check functions

This value can be used for the metrics parameter in the check functions, i.e. width_check. This value specifies Euclidian metrics, i.e. the distance between two points is measured by:

```
d = sqrt(dx^2 + dy^2)
```

All points within a circle with radius d around one point are considered to have a smaller distance than d.

Python specific notes:
The object exposes a readable attribute 'Euclidian'. This is the getter.

Projection

Signature: *[static,const]* Metrics **Projection**

Description: Specifies projected distance metrics for the check functions

This value can be used for the metrics parameter in the check functions, i.e. width_check. This value specifies projected metrics, i.e. the distance is defined as the minimum distance measured perpendicular to one edge. That implies that the distance is defined only where two edges have a non-vanishing projection onto each other.

Python specific notes:
The object exposes a readable attribute 'Projection'. This is the getter.

Square

Signature: *[static,const]* Metrics **Square**

Description: Specifies square metrics for the check functions

This value can be used for the metrics parameter in the check functions, i.e. width_check. This value specifies square metrics, i.e. the distance between two points is measured by:

```
d = max(abs(dx), abs(dy))
```

All points within a square with length 2*d around one point are considered to have a smaller distance than d in this metrics.

Python specific notes:
The object exposes a readable attribute 'Square'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Metrics other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Metrics ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

(1) Signature: *[static]* new Metrics ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Metrics ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.91. API reference - Class EdgeMode

Notation used in Ruby API documentation

Module: db

Description: This class represents the edge mode type for \Region#edges.

This enum has been introduced in version 0.29.

Public constructors

new EdgeMode ptr	**new**	(int i)	Creates an enum from an integer value
new EdgeMode ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const EdgeMode other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const EdgeMode other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const EdgeMode other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const EdgeMode other)	Assigns another object to self
[const]	new EdgeMode ptr	**dup**		Creates a copy of self

[const]	int	**hash**	Gets the hash value from the enum
[const]	string	**inspect**	Converts an enum to a visual string
[const]	int	**to_i**	Gets the integer value from the enum
[const]	string	**to_s**	Gets the symbolic string from an enum

Public static methods and constants

[static,const]	EdgeMode	**All**	Selects all edges
[static,const]	EdgeMode	**Concave**	Selects only concave edges
[static,const]	EdgeMode	**Convex**	Selects only convex edges
[static,const]	EdgeMode	**NotConcave**	Selects only edges which are not concave
[static,const]	EdgeMode	**NotConvex**	Selects only edges which are not convex
[static,const]	EdgeMode	**NotStep**	Selects only edges which are not steps
[static,const]	EdgeMode	**NotStepIn**	Selects only edges which are not steps leading inside
[static,const]	EdgeMode	**NotStepOut**	Selects only edges which are not steps leading outside
[static,const]	EdgeMode	**Step**	Selects only step edges leading inside or outside
[static,const]	EdgeMode	**StepIn**	Selects only step edges leading inside
[static,const]	EdgeMode	**StepOut**	Selects only step edges leading outside

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

(1) Signature: *[const]* bool **!=** (const EdgeMode other)
Description: Compares two enums for inequality

(2) Signature: *[const]* bool **!=** (int other)

	Description: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const EdgeMode other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const EdgeMode other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
All	**Signature**: *[static,const]* EdgeMode **All** **Description**: Selects all edges **Python specific notes:** The object exposes a readable attribute 'All'. This is the getter.
Concave	**Signature**: *[static,const]* EdgeMode **Concave** **Description**: Selects only concave edges **Python specific notes:** The object exposes a readable attribute 'Concave'. This is the getter.
Convex	**Signature**: *[static,const]* EdgeMode **Convex** **Description**: Selects only convex edges **Python specific notes:** The object exposes a readable attribute 'Convex'. This is the getter.
NotConcave	**Signature**: *[static,const]* EdgeMode **NotConcave** **Description**: Selects only edges which are not concave **Python specific notes:** The object exposes a readable attribute 'NotConcave'. This is the getter.
NotConvex	**Signature**: *[static,const]* EdgeMode **NotConvex** **Description**: Selects only edges which are not convex **Python specific notes:** The object exposes a readable attribute 'NotConvex'. This is the getter.
NotStep	**Signature**: *[static,const]* EdgeMode **NotStep** **Description**: Selects only edges which are not steps **Python specific notes:** The object exposes a readable attribute 'NotStep'. This is the getter.
NotStepIn	**Signature**: *[static,const]* EdgeMode **NotStepIn** **Description**: Selects only edges which are not steps leading inside

Python specific notes:
The object exposes a readable attribute 'NotStepIn'. This is the getter.

NotStepOut

Signature: *[static,const]* EdgeMode **NotStepOut**

Description: Selects only edges which are not steps leading outside

Python specific notes:
The object exposes a readable attribute 'NotStepOut'. This is the getter.

Step

Signature: *[static,const]* EdgeMode **Step**

Description: Selects only step edges leading inside or outside

Python specific notes:
The object exposes a readable attribute 'Step'. This is the getter.

StepIn

Signature: *[static,const]* EdgeMode **StepIn**

Description: Selects only step edges leading inside

Python specific notes:
The object exposes a readable attribute 'StepIn'. This is the getter.

StepOut

Signature: *[static,const]* EdgeMode **StepOut**

Description: Selects only step edges leading outside

Python specific notes:
The object exposes a readable attribute 'StepOut'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const EdgeMode other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new EdgeMode ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:

This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

(1) Signature: *[static]* new EdgeMode ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new EdgeMode ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.92. API reference - Class ZeroDistanceMode

Notation used in Ruby API documentation

Module: db

Description: This class represents the zero_distance_mode type for \Region#width and related checks.

This mode determines how edges with zero distance are treated in the DRC checks. Formally these edges do neither represent a space other other relation as they do not face each other. There are three modes available to treat this boundary case: Ignore such edges (NeverIncludeZeroDistance) or only include them if they share at least one common point (IncludeZeroDistanceWhenTouching). The latter mode allows activating checks for the 'kissing corner' case and is the default mode in most checks. This enum has been introduced in version 0.28.16.

Public constructors

new ZeroDistanceMode ptr	**new**	(int i)	Creates an enum from an integer value
new ZeroDistanceMode ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const ZeroDistanceMode other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const ZeroDistanceMode other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const ZeroDistanceMode other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

	void	**assign**	(const ZeroDistanceMode other)	Assigns another object to self
[const]	new ZeroDistanceMode ptr	**dup**		Creates a copy of self
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	ZeroDistanceMode	**IncludeZeroDistanceWhenC**	Specifies that check functions should include edges when they are collinear and touch
[static,const]	ZeroDistanceMode	**IncludeZeroDistanceWhenOverlapping**	Specifies that check functions should include edges when they overlap
[static,const]	ZeroDistanceMode	**IncludeZeroDistanceWhenT**	Specifies that check functions should include edges when they touch
[static,const]	ZeroDistanceMode	**NeverIncludeZeroDistance**	Specifies that check functions should never include edges with zero distance.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const ZeroDistanceMode other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const ZeroDistanceMode other)

Description: Returns true if the first enum is less (in the enum symbol order) than the second

(2) Signature: *[const]* bool **<** (int other)

Description: Returns true if the enum is less (in the enum symbol order) than the integer value

==

(1) Signature: *[const]* bool **==** (const ZeroDistanceMode other)

Description: Compares two enums

(2) Signature: *[const]* bool **==** (int other)

Description: Compares an enum with an integer value

IncludeZeroDistanceWhenCollinearAndTouching

Signature: *[static,const]* ZeroDistanceMode **IncludeZeroDistanceWhenCollinearAndTouching**

Description: Specifies that check functions should include edges when they are collinear and touch

With this specification, the check functions will also check edges if they share at least one common point and are collinear. This is the mode that includes checking the 'kissing corner' cases when the kissing edges are collinear. This mode was default up to version 0.28.

Python specific notes:
The object exposes a readable attribute 'IncludeZeroDistanceWhenCollinearAndTouching'. This is the getter.

IncludeZeroDistanceWhenO

Signature: *[static,const]* ZeroDistanceMode **IncludeZeroDistanceWhenOverlapping**

Description: Specifies that check functions should include edges when they overlap

With this specification, the check functions will also check edges which are collinear and share more than a single point. This is the mode that excludes the 'kissing corner' cases.

Python specific notes:
The object exposes a readable attribute 'IncludeZeroDistanceWhenOverlapping'. This is the getter.

IncludeZeroDistanceWhenTouching

Signature: *[static,const]* ZeroDistanceMode **IncludeZeroDistanceWhenTouching**

Description: Specifies that check functions should include edges when they touch

With this specification, the check functions will also check edges if they share at least one common point. This is the mode that includes checking the 'kissing corner' cases. This mode is default for version 0.28.16 and later.

Python specific notes:
The object exposes a readable attribute 'IncludeZeroDistanceWhenTouching'. This is the getter.

NeverIncludeZeroDistance

Signature: *[static,const]* ZeroDistanceMode **NeverIncludeZeroDistance**

Description: Specifies that check functions should never include edges with zero distance.

With this specification, the check functions will ignore edges which are collinear or touch.

Python specific notes:
The object exposes a readable attribute 'NeverIncludeZeroDistance'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const ZeroDistanceMode other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new ZeroDistanceMode ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

(1) Signature: *[static]* new ZeroDistanceMode ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new ZeroDistanceMode ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.93. API reference - Class PropertyConstraint

Notation used in Ruby API documentation

Module: db

Description: This class represents the property constraint for boolean and check functions.

This enum has been introduced in version 0.28.4.

Public constructors

new PropertyConstraint ptr	**new**	(int i)	Creates an enum from an integer value
new PropertyConstraint ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const PropertyConstrain other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const PropertyConstrain other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const PropertyConstrain other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const PropertyConstrain other)	Assigns another object to self

[const]	new PropertyConstraint ptr	**dup**	Creates a copy of self
[const]	int	**hash**	Gets the hash value from the enum
[const]	string	**inspect**	Converts an enum to a visual string
[const]	int	**to_i**	Gets the integer value from the enum
[const]	string	**to_s**	Gets the symbolic string from an enum

Public static methods and constants

[static,const]	PropertyConstraint	**DifferentPropertiesConstrai**	Specifies to consider shapes only if their user properties are different
[static,const]	PropertyConstraint	**DifferentPropertiesConstraintDrop**	Specifies to consider shapes only if their user properties are different
[static,const]	PropertyConstraint	**IgnoreProperties**	Specifies to ignore properties
[static,const]	PropertyConstraint	**NoPropertyConstraint**	Specifies not to apply any property constraint
[static,const]	PropertyConstraint	**SamePropertiesConstraint**	Specifies to consider shapes only if their user properties are the same
[static,const]	PropertyConstraint	**SamePropertiesConstraintDrop**	Specifies to consider shapes only if their user properties are the same

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const PropertyConstraint other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const PropertyConstraint other)

Description: Returns true if the first enum is less (in the enum symbol order) than the second

(2) Signature: *[const]* bool **<** (int other)

Description: Returns true if the enum is less (in the enum symbol order) than the integer value

==

(1) Signature: *[const]* bool **==** (const PropertyConstraint other)

Description: Compares two enums

(2) Signature: *[const]* bool **==** (int other)

Description: Compares an enum with an integer value

DifferentPropertiesConstraint

Signature: *[static,const]* PropertyConstraint **DifferentPropertiesConstraint**

Description: Specifies to consider shapes only if their user properties are different

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are different. Properties are generated on the output shapes where applicable.

Python specific notes:
The object exposes a readable attribute 'DifferentPropertiesConstraint'. This is the getter.

DifferentPropertiesConstrai

Signature: *[static,const]* PropertyConstraint **DifferentPropertiesConstraintDrop**

Description: Specifies to consider shapes only if their user properties are different

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are the same. No properties are generated on the output shapes.

Python specific notes:
The object exposes a readable attribute 'DifferentPropertiesConstraintDrop'. This is the getter.

IgnoreProperties

Signature: *[static,const]* PropertyConstraint **IgnoreProperties**

Description: Specifies to ignore properties

When using this constraint - for example on a boolean operation - properties are ignored and are not generated in the output.

Python specific notes:
The object exposes a readable attribute 'IgnoreProperties'. This is the getter.

NoPropertyConstraint

Signature: *[static,const]* PropertyConstraint **NoPropertyConstraint**

Description: Specifies not to apply any property constraint

When using this constraint - for example on a boolean operation - shapes are considered regardless of their user properties. Properties are generated on the output shapes where applicable.

Python specific notes:
The object exposes a readable attribute 'NoPropertyConstraint'. This is the getter.

SamePropertiesConstraint

Signature: *[static,const]* PropertyConstraint **SamePropertiesConstraint**

Description: Specifies to consider shapes only if their user properties are the same

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are the same. Properties are generated on the output shapes where applicable.

Python specific notes:
The object exposes a readable attribute 'SamePropertiesConstraint'. This is the getter.

SamePropertiesConstraintD

Signature: *[static,const]* PropertyConstraint **SamePropertiesConstraintDrop**

Description: Specifies to consider shapes only if their user properties are the same

When using this constraint - for example on a boolean operation - shapes are considered only if their user properties are the same. No properties are generated on the output shapes.

Python specific notes:
The object exposes a readable attribute 'SamePropertiesConstraintDrop'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const PropertyConstraint other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new PropertyConstraint ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

(1) Signature: *[static]* new PropertyConstraint ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new PropertyConstraint ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.94. API reference - Class Shape

Notation used in Ruby API documentation

Module: db

Description: An object representing a shape in the layout database

The shape proxy is basically a pointer to a shape of different kinds. No copy of the shape is created: if the shape proxy is copied the copy still points to the original shape. If the original shape is modified or deleted, the shape proxy will also point to a modified or invalid shape. The proxy can be "null" which indicates an invalid reference.

Shape objects are used together with the Shapes container object which stores the actual shape objects and uses Shape references as pointers inside the actual data storage. Shape references are used in various places, i.e. when removing or transforming objects inside a Shapes container.

Public constructors

new Shape ptr	**new**		Creates a new object of this class

Public methods

[const]	bool	**!=**	(const Shape other)	Inequality operator
[const]	bool	**==**	(const Shape other)	Equality operator
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	long	**area**		Returns the area of the shape
[const]	DTrans	**array_dtrans**		Gets the array instance member transformation in micrometer units
[const]	Trans	**array_trans**		Gets the array instance member transformation
	void	**assign**	(const Shape other)	Assigns another object to self
[const]	Box	**bbox**		Returns the bounding box of the shape
[const]	variant	**box**		Gets the box object

	void	**box=**	(const Box box)	Replaces the shape by the given box
	void	**box=**	(const DBox box)	Replaces the shape by the given box (in micrometer units)
[const]	Point	**box_center**		Returns the center of the box
	void	**box_center=**	(const Point c)	Sets the center of the box
	void	**box_center=**	(const DPoint c)	Sets the center of the box with the point being given in micrometer units
[const]	DPoint	**box_dcenter**		Returns the center of the box as a DPoint object in micrometer units
	void	**box_dcenter=**	(const DPoint c)	Sets the center of the box with the point being given in micrometer units
[const]	double	**box_dheight**		Returns the height of the box in micrometer units
	void	**box_dheight=**	(double h)	Sets the height of the box
[const]	DPoint	**box_dp1**		Returns the lower left point of the box as a DPoint object in micrometer units
	void	**box_dp1=**	(const DPoint p)	Sets the lower left corner of the box with the point being given in micrometer units
[const]	DPoint	**box_dp2**		Returns the upper right point of the box as a DPoint object in micrometer units
	void	**box_dp2=**	(const DPoint p)	Sets the upper right corner of the box with the point being given in micrometer units
[const]	double	**box_dwidth**		Returns the width of the box in micrometer units
	void	**box_dwidth=**	(double w)	Sets the width of the box in micrometer units
[const]	int	**box_height**		Returns the height of the box
	void	**box_height=**	(int h)	Sets the height of the box
[const]	Point	**box_p1**		Returns the lower left point of the box
	void	**box_p1=**	(const Point p)	Sets the lower left point of the box
	void	**box_p1=**	(const DPoint p)	Sets the lower left corner of the box with the point being given in micrometer units
[const]	Point	**box_p2**		Returns the upper right point of the box
	void	**box_p2=**	(const Point p)	Sets the upper right point of the box
	void	**box_p2=**	(const DPoint p)	Sets the upper right corner of the box with the point being given in micrometer units

[const]	int	**box_width**		Returns the width of the box
	void	**box_width=**	(int w)	Sets the width of the box
	Cell ptr	**cell**		Gets a reference to the cell the shape belongs to
	void	**cell=**	(Cell ptr cell)	Moves the shape to a different cell
[const]	double	**darea**		Returns the area of the shape in square micrometer units
[const]	DBox	**dbbox**		Returns the bounding box of the shape in micrometer units
[const]	variant	**dbox**		Gets the box object in micrometer units
	void	**dbox=**	(const DBox box)	Replaces the shape by the given box (in micrometer units)
[const]	variant	**dedge**		Returns the edge object as a DEdge object in micrometer units
	void	**dedge=**	(const DEdge edge)	Replaces the shape by the given edge (in micrometer units)
[const]	variant	**dedge_pair**		Returns the edge pair object as a DEdgePair object in micrometer units
	void	**dedge_pair=**	(const DEdgePair edge_pair)	Replaces the shape by the given edge pair (in micrometer units)
	void	**delete**		Deletes the shape
	void	**delete_property**	(variant key)	Deletes the user property with the given key
[const]	variant	**dpath**		Returns the path object as a DPath object in micrometer units
	void	**dpath=**	(const DPath path)	Replaces the shape by the given path (in micrometer units)
[const]	double	**dperimeter**		Returns the perimeter of the shape in micrometer units
[const]	variant	**dpoint**		Returns the point object as a DPoint object in micrometer units
	void	**dpoint=**	(const DPoint point)	Replaces the shape by the given point (in micrometer units)
[const]	variant	**dpolygon**		Returns the polygon object in micrometer units
	void	**dpolygon=**	(const DPolygon polygon)	Replaces the shape by the given polygon (in micrometer units)

[const]	variant	**drectangle**		Gets the rectangle in micron units if the object represents one or nil if not
[const]	variant	**dsimple_polygon**		Returns the simple polygon object in micrometer units
	void	**dsimple_polygon=**	(const DSimplePolygon polygon)	Replaces the shape by the given simple polygon (in micrometer units)
[const]	variant	**dtext**		Returns the path object as a DText object in micrometer units
	void	**dtext=**	(const DText text)	Replaces the shape by the given text (in micrometer units)
[const]	new Shape ptr	**dup**		Creates a copy of self
[const,iter]	DEdge	**each_dedge**		Iterates over the edges of the object and returns edges in micrometer units
[const,iter]	DEdge	**each_dedge**	(unsigned int contour)	Iterates over the edges of a single contour of the object and returns edges in micrometer units
[const,iter]	DPoint	**each_dpoint**		Iterates over all points of the object and returns points in micrometer units
[const,iter]	DPoint	**each_dpoint_hole**	(unsigned int hole_index)	Iterates over a hole contour of the object and returns points in micrometer units
[const,iter]	DPoint	**each_dpoint_hull**		Iterates over the hull contour of the object and returns points in micrometer units
[const,iter]	Edge	**each_edge**		Iterates over the edges of the object
[const,iter]	Edge	**each_edge**	(unsigned int contour)	Iterates over the edges of a single contour of the object
[const,iter]	Point	**each_point**		Iterates over all points of the object
[const,iter]	Point	**each_point_hole**	(unsigned int hole_index)	Iterates over the points of a hole contour
[const,iter]	Point	**each_point_hull**		Iterates over the hull contour of the object
[const]	variant	**edge**		Returns the edge object
	void	**edge=**	(const Edge edge)	Replaces the shape by the given edge
	void	**edge=**	(const DEdge edge)	Replaces the shape by the given edge (in micrometer units)
[const]	variant	**edge_pair**		Returns the edge pair object
	void	**edge_pair=**	(const EdgePair edge_pair)	Replaces the shape by the given edge pair

	void	**edge_pair=**	(const DEdgePair edge_pair)	Replaces the shape by the given edge pair (in micrometer units)
[const]	bool	**has_prop_id?**		Returns true, if the shape has properties, i.e. has a properties ID
[const]	unsigned int	**holes**		Returns the number of holes
[const]	bool	**is_array_member?**		Returns true, if the shape is a member of a shape array
[const]	bool	**is_box?**		Returns true if the shape is a box
[const]	bool	**is_edge?**		Returns true, if the object is an edge
[const]	bool	**is_edge_pair?**		Returns true, if the object is an edge pair
[const]	bool	**is_null?**		Returns true, if the shape reference is a null reference (not referring to a shape)
[const]	bool	**is_path?**		Returns true, if the shape is a path
[const]	bool	**is_point?**		Returns true, if the object is an point
[const]	bool	**is_polygon?**		Returns true, if the shape is a polygon
[const]	bool	**is_simple_polygon?**		Returns true, if the shape is a simple polygon
[const]	bool	**is_text?**		Returns true, if the object is a text
[const]	bool	**is_user_object?**		Returns true if the shape is a user defined object
[const]	bool	**is_valid?**		Returns true, if the shape is valid
[const]	unsigned int	**layer**		Returns the layer index of the layer the shape is on
	void	**layer=**	(unsigned int layer_index)	Moves the shape to a layer given by the layer index object
[const]	LayerInfo	**layer_info**		Returns the LayerInfo object of the layer the shape is on
	void	**layer_info=**	(const LayerInfo layer_info)	Moves the shape to a layer given by a LayerInfo object
	Layout ptr	**layout**		Gets a reference to the Layout the shape belongs to
[const]	variant	**path**		Returns the path object
	void	**path=**	(const Path box)	Replaces the shape by the given path object

	void	**path=**	(const DPath path)	Replaces the shape by the given path (in micrometer units)
[const]	int	**path_bgnext**		Gets the path's starting vertex extension
	void	**path_bgnext=**	(int e)	Sets the path's starting vertex extension
[const]	double	**path_dbgnext**		Gets the path's starting vertex extension in micrometer units
	void	**path_dbgnext=**	(double e)	Sets the path's starting vertex extension in micrometer units
[const]	double	**path_dendext**		Gets the path's end vertex extension in micrometer units
	void	**path_dendext=**	(double e)	Sets the path's end vertex extension in micrometer units
[const]	double	**path_dlength**		Returns the length of the path in micrometer units
[const]	double	**path_dwidth**		Gets the path width in micrometer units
	void	**path_dwidth=**	(double w)	Sets the path width in micrometer units
[const]	int	**path_endext**		Obtain the path's end vertex extension
	void	**path_endext=**	(int e)	Sets the path's end vertex extension
[const]	int	**path_length**		Returns the length of the path
[const]	int	**path_width**		Gets the path width
	void	**path_width=**	(int w)	Sets the path width
[const]	unsigned long	**perimeter**		Returns the perimeter of the shape
[const]	variant	**point**		Returns the point object
	void	**point=**	(const Point point)	Replaces the shape by the given point
	void	**point=**	(const DPoint point)	Replaces the shape by the given point (in micrometer units)
[const]	variant	**polygon**		Returns the polygon object
	void	**polygon=**	(const Polygon box)	Replaces the shape by the given polygon object
	void	**polygon=**	(const DPolygon polygon)	Replaces the shape by the given polygon (in micrometer units)
[const]	unsigned long	**prop_id**		Gets the properties ID associated with the shape

	void	**prop_id=**	(unsigned long id)	Sets the properties ID of this shape
[const]	variant	**property**	(variant key)	Gets the user property with the given key
[const]	variant	**rectangle**		Gets the rectangle if the object represents one or nil if not
	void	**round_path=**	(bool r)	The path will be a round-ended path if this property is set to true
[const]	bool	**round_path?**		Returns true, if the path has round ends
	void	**set_property**	(variant key, variant value)	Sets the user property with the given key to the given value
	Shapes ptr	**shapes**		Gets a reference to the Shapes container the shape lives in
[const]	variant	**simple_polygon**		Returns the simple polygon object
	void	**simple_polygon=**	(const SimplePolygon polygon)	Replaces the shape by the given simple polygon object
	void	**simple_polygon=**	(const DSimplePolygon polygon)	Replaces the shape by the given simple polygon (in micrometer units)
[const]	variant	**text**		Returns the text object
	void	**text=**	(const Text box)	Replaces the shape by the given text object
	void	**text=**	(const DText text)	Replaces the shape by the given text (in micrometer units)
[const]	DVector	**text_dpos**		Gets the text's position in micrometer units
	void	**text_dpos=**	(const DVector p)	Sets the text's position in micrometer units
[const]	double	**text_dsize**		Gets the text size in micrometer units
	void	**text_dsize=**	(double size)	Sets the text size in micrometer units
[const]	DTrans	**text_dtrans**		Gets the text transformation in micrometer units
	void	**text_dtrans=**	(const DTrans trans)	Sets the text transformation in micrometer units
[const]	int	**text_font**		Gets the text's font
	void	**text_font=**	(int font)	Sets the text's font
[const]	int	**text_halign**		Gets the text's horizontal alignment
	void	**text_halign=**	(int a)	Sets the text's horizontal alignment

[const]	Vector	**text_pos**		Gets the text's position
	void	**text_pos=**	(const Vector p)	Sets the text's position
	void	**text_pos=**	(const DVector p)	Sets the text's position in micrometer units
[const]	int	**text_rot**		Gets the text's orientation code (see Trans)
	void	**text_rot=**	(int o)	Sets the text's orientation code (see Trans)
[const]	int	**text_size**		Gets the text size
	void	**text_size=**	(int size)	Sets the text size
[const]	string	**text_string**		Obtain the text string
	void	**text_string=**	(string string)	Sets the text string
[const]	Trans	**text_trans**		Gets the text transformation
	void	**text_trans=**	(const Trans trans)	Sets the text transformation
	void	**text_trans=**	(const DTrans trans)	Sets the text transformation in micrometer units
[const]	int	**text_valign**		Gets the text's vertical alignment
	void	**text_valign=**	(int a)	Sets the text's vertical alignment
[const]	string	**to_s**		Create a string showing the contents of the reference
	void	**transform**	(const Trans trans)	Transforms the shape with the given transformation
	void	**transform**	(const DTrans trans)	Transforms the shape with the given transformation, given in micrometer units
	void	**transform**	(const ICplxTrans trans)	Transforms the shape with the given complex transformation
	void	**transform**	(const DCplxTrans trans)	Transforms the shape with the given complex transformation, given in micrometer units
[const]	int	**type**		Return the type of the shape

Public static methods and constants

	int	**TBox**
	int	**TBoxArray**

int	TBoxArrayMember
int	TEdge
int	TEdgePair
int	TNull
int	TPath
int	TPathPtrArray
int	TPathPtrArrayMember
int	TPathRef
int	TPoint
int	TPolygon
int	TPolygonPtrArray
int	TPolygonPtrArrayMember
int	TPolygonRef
int	TShortBox
int	TShortBoxArray
int	TShortBoxArrayMember
int	TSimplePolygon
int	TSimplePolygonPtrArray
int	TSimplePolygonPtrArrayMember
int	TSimplePolygonRef
int	TText
int	TTextPtrArray
int	TTextPtrArrayMember
int	TTextRef
int	TUserObject

Deprecated methods (protected, public, static, non-static and constructors)

void	create	Use of this method is deprecated. Use _create instead
void	destroy	Use of this method is deprecated. Use _destroy instead

[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead
[static]	int	**t_box**	Use of this method is deprecated. Use TBox instead
[static]	int	**t_box_array**	Use of this method is deprecated. Use TBoxArray instead
[static]	int	**t_box_array_member**	Use of this method is deprecated. Use TBoxArrayMember instead
[static]	int	**t_edge**	Use of this method is deprecated. Use TEdge instead
[static]	int	**t_edge_pair**	Use of this method is deprecated. Use TEdgePair instead
[static]	int	**t_null**	Use of this method is deprecated. Use TNull instead
[static]	int	**t_path**	Use of this method is deprecated. Use TPath instead
[static]	int	**t_path_ptr_array**	Use of this method is deprecated. Use TPathPtrArray instead
[static]	int	**t_path_ptr_array_member**	Use of this method is deprecated. Use TPathPtrArrayMember instead
[static]	int	**t_path_ref**	Use of this method is deprecated. Use TPathRef instead
[static]	int	**t_point**	Use of this method is deprecated. Use TPoint instead
[static]	int	**t_polygon**	Use of this method is deprecated. Use TPolygon instead
[static]	int	**t_polygon_ptr_array**	Use of this method is deprecated. Use TPolygonPtrArray instead
[static]	int	**t_polygon_ptr_array_member**	Use of this method is deprecated. Use TPolygonPtrArrayMember instead
[static]	int	**t_polygon_ref**	Use of this method is deprecated. Use TPolygonRef instead
[static]	int	**t_short_box**	Use of this method is deprecated. Use TShortBox instead
[static]	int	**t_short_box_array**	Use of this method is deprecated. Use TShortBoxArray instead
[static]	int	**t_short_box_array_member**	Use of this method is deprecated. Use TShortBoxArrayMember instead
[static]	int	**t_simple_polygon**	Use of this method is deprecated. Use TSimplePolygon instead

[static]	int	**t_simple_polygon_ptr_array**	Use of this method is deprecated. Use TSimplePolygonPtrArray instead
[static]	int	**t_simple_polygon_ptr_array_member**	Use of this method is deprecated. Use TSimplePolygonPtrArrayMember instead
[static]	int	**t_simple_polygon_ref**	Use of this method is deprecated. Use TSimplePolygonRef instead
[static]	int	**t_text**	Use of this method is deprecated. Use TText instead
[static]	int	**t_text_ptr_array**	Use of this method is deprecated. Use TTextPtrArray instead
[static]	int	**t_text_ptr_array_member**	Use of this method is deprecated. Use TTextPtrArrayMember instead
[static]	int	**t_text_ref**	Use of this method is deprecated. Use TTextRef instead
[static]	int	**t_user_object**	Use of this method is deprecated. Use TUserObject instead

Detailed description

!=

Signature: *[const]* bool **!=** (const Shape other)

Description: Inequality operator

==

Signature: *[const]* bool **==** (const Shape other)

Description: Equality operator

Equality of shapes is not specified by the identity of the objects but by the identity of the pointers - both shapes must refer to the same object.

TBox

Signature: *[static]* int **TBox**

Description:

Python specific notes:
The object exposes a readable attribute 'TBox'. This is the getter.

TBoxArray

Signature: *[static]* int **TBoxArray**

Description:

Python specific notes:
The object exposes a readable attribute 'TBoxArray'. This is the getter.

TBoxArrayMember

Signature: *[static]* int **TBoxArrayMember**

Description:

Python specific notes:
The object exposes a readable attribute 'TBoxArrayMember'. This is the getter.

TEdge

Signature: *[static]* int **TEdge**

Description:

Python specific notes:
The object exposes a readable attribute 'TEdge'. This is the getter.

TEdgePair

Signature: *[static]* int **TEdgePair**

Description:

Python specific notes:
The object exposes a readable attribute 'TEdgePair'. This is the getter.

TNull

Signature: *[static]* int **TNull**

Description:

Python specific notes:
The object exposes a readable attribute 'TNull'. This is the getter.

TPath

Signature: *[static]* int **TPath**

Description:

Python specific notes:
The object exposes a readable attribute 'TPath'. This is the getter.

TPathPtrArray

Signature: *[static]* int **TPathPtrArray**

Description:

Python specific notes:
The object exposes a readable attribute 'TPathPtrArray'. This is the getter.

TPathPtrArrayMember

Signature: *[static]* int **TPathPtrArrayMember**

Description:

Python specific notes:
The object exposes a readable attribute 'TPathPtrArrayMember'. This is the getter.

TPathRef

Signature: *[static]* int **TPathRef**

Description:

Python specific notes:
The object exposes a readable attribute 'TPathRef'. This is the getter.

TPoint

Signature: *[static]* int **TPoint**

Description:

Python specific notes:
The object exposes a readable attribute 'TPoint'. This is the getter.

TPolygon

Signature: *[static]* int **TPolygon**

Description:

Python specific notes:
The object exposes a readable attribute 'TPolygon'. This is the getter.

TPolygonPtrArray

Signature: *[static]* int **TPolygonPtrArray**

Description:

Python specific notes:

	The object exposes a readable attribute 'TPolygonPtrArray'. This is the getter.
TPolygonPtrArrayMember	**Signature**: *[static]* int **TPolygonPtrArrayMember** **Description**: **Python specific notes:** The object exposes a readable attribute 'TPolygonPtrArrayMember'. This is the getter.
TPolygonRef	**Signature**: *[static]* int **TPolygonRef** **Description**: **Python specific notes:** The object exposes a readable attribute 'TPolygonRef'. This is the getter.
TShortBox	**Signature**: *[static]* int **TShortBox** **Description**: **Python specific notes:** The object exposes a readable attribute 'TShortBox'. This is the getter.
TShortBoxArray	**Signature**: *[static]* int **TShortBoxArray** **Description**: **Python specific notes:** The object exposes a readable attribute 'TShortBoxArray'. This is the getter.
TShortBoxArrayMember	**Signature**: *[static]* int **TShortBoxArrayMember** **Description**: **Python specific notes:** The object exposes a readable attribute 'TShortBoxArrayMember'. This is the getter.
TSimplePolygon	**Signature**: *[static]* int **TSimplePolygon** **Description**: **Python specific notes:** The object exposes a readable attribute 'TSimplePolygon'. This is the getter.
TSimplePolygonPtrArray	**Signature**: *[static]* int **TSimplePolygonPtrArray** **Description**: **Python specific notes:** The object exposes a readable attribute 'TSimplePolygonPtrArray'. This is the getter.
TSimplePolygonPtrArrayM	**Signature**: *[static]* int **TSimplePolygonPtrArrayMember** **Description**: **Python specific notes:** The object exposes a readable attribute 'TSimplePolygonPtrArrayMember'. This is the getter.
TSimplePolygonRef	**Signature**: *[static]* int **TSimplePolygonRef** **Description**: **Python specific notes:** The object exposes a readable attribute 'TSimplePolygonRef'. This is the getter.

TText

Signature: *[static]* int **TText**

Description:

Python specific notes:
The object exposes a readable attribute 'TText'. This is the getter.

TTextPtrArray

Signature: *[static]* int **TTextPtrArray**

Description:

Python specific notes:
The object exposes a readable attribute 'TTextPtrArray'. This is the getter.

TTextPtrArrayMember

Signature: *[static]* int **TTextPtrArrayMember**

Description:

Python specific notes:
The object exposes a readable attribute 'TTextPtrArrayMember'. This is the getter.

TTextRef

Signature: *[static]* int **TTextRef**

Description:

Python specific notes:
The object exposes a readable attribute 'TTextRef'. This is the getter.

TUserObject

Signature: *[static]* int **TUserObject**

Description:

Python specific notes:
The object exposes a readable attribute 'TUserObject'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

area

Signature: *[const]* long **area**

Description: Returns the area of the shape

This method has been added in version 0.22.

array_dtrans

Signature: *[const]* DTrans **array_dtrans**

Description: Gets the array instance member transformation in micrometer units

This attribute is valid only if is_array_member? is true. The transformation returned describes the relative transformation of the array member addressed. The displacement is given in micrometer units.

This method has been added in version 0.25.

array_trans

Signature: *[const]* Trans **array_trans**

Description: Gets the array instance member transformation

This attribute is valid only if is_array_member? is true. The transformation returned describes the relative transformation of the array member addressed.

assign

Signature: void **assign** (const Shape other)

Description: Assigns another object to self

bbox

Signature: *[const]* Box **bbox**

Description: Returns the bounding box of the shape

box

Signature: *[const]* variant **box**

Description: Gets the box object

Starting with version 0.23, this method returns nil, if the shape does not represent a box.

Python specific notes:
The object exposes a readable attribute 'box'. This is the getter.

box=

(1) Signature: void **box=** (const Box box)

Description: Replaces the shape by the given box

This method replaces the shape by the given box. This method can only be called for editable layouts. It does not change the user properties of the shape. Calling this method will invalidate any iterators. It should not be called inside a loop iterating over shapes.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'box'. This is the setter.

(2) Signature: void **box=** (const DBox box)

Description: Replaces the shape by the given box (in micrometer units)

This method replaces the shape by the given box, like box= with a Box argument does. This version translates the box from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'box'. This is the setter.
The object exposes a writable attribute 'dbox'. This is the setter.

box_center

Signature: *[const]* Point **box_center**

Description: Returns the center of the box

Applies to boxes only. Returns the center of the box and throws an exception if the shape is not a box.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'box_center'. This is the getter.

box_center=

(1) Signature: void **box_center=** (const Point c)

Description: Sets the center of the box

Applies to boxes only. Changes the center of the box and throws an exception if the shape is not a box.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'box_center'. This is the setter.

(2) Signature: void **box_center=** (const DPoint c)

Description: Sets the center of the box with the point being given in micrometer units

Applies to boxes only. Changes the center of the box and throws an exception if the shape is not a box. Translation from micrometer units to database units is done internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'box_center'. This is the setter.
The object exposes a writable attribute 'box_dcenter'. This is the setter.

box_dcenter

Signature: *[const]* DPoint **box_dcenter**

Description: Returns the center of the box as a DPoint object in micrometer units

Applies to boxes only. Returns the center of the box and throws an exception if the shape is not a box. Conversion from database units to micrometers is done internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'box_dcenter'. This is the getter.

box_dcenter=

Signature: void **box_dcenter=** (const DPoint c)

Description: Sets the center of the box with the point being given in micrometer units

Applies to boxes only. Changes the center of the box and throws an exception if the shape is not a box. Translation from micrometer units to database units is done internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'box_center'. This is the setter.
The object exposes a writable attribute 'box_dcenter'. This is the setter.

box_dheight

Signature: *[const]* double **box_dheight**

Description: Returns the height of the box in micrometer units

Applies to boxes only. Returns the height of the box in micrometers and throws an exception if the shape is not a box.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'box_dheight'. This is the getter.

box_dheight=

Signature: void **box_dheight=** (double h)

Description: Sets the height of the box

Applies to boxes only. Changes the height of the box to the value given in micrometer units and throws an exception if the shape is not a box. Translation to database units happens internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'box_dheight'. This is the setter.

box_dp1

Signature: *[const]* DPoint **box_dp1**

Description: Returns the lower left point of the box as a DPoint object in micrometer units

Applies to boxes only. Returns the lower left point of the box and throws an exception if the shape is not a box. Conversion from database units to micrometers is done internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'box_dp1'. This is the getter.

box_dp1=

Signature: void **box_dp1=** (const DPoint p)

Description: Sets the lower left corner of the box with the point being given in micrometer units

Applies to boxes only. Changes the lower left point of the box and throws an exception if the shape is not a box. Translation from micrometer units to database units is done internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'box_p1'. This is the setter.

The object exposes a writable attribute 'box_dp1'. This is the setter.

box_dp2

Signature: *[const]* DPoint **box_dp2**

Description: Returns the upper right point of the box as a DPoint object in micrometer units

Applies to boxes only. Returns the upper right point of the box and throws an exception if the shape is not a box. Conversion from database units to micrometers is done internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'box_dp2'. This is the getter.

box_dp2=

Signature: void **box_dp2=** (const DPoint p)

Description: Sets the upper right corner of the box with the point being given in micrometer units

Applies to boxes only. Changes the upper right point of the box and throws an exception if the shape is not a box. Translation from micrometer units to database units is done internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'box_p2'. This is the setter.
The object exposes a writable attribute 'box_dp2'. This is the setter.

box_dwidth

Signature: *[const]* double **box_dwidth**

Description: Returns the width of the box in micrometer units

Applies to boxes only. Returns the width of the box in micrometers and throws an exception if the shape is not a box.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'box_dwidth'. This is the getter.

box_dwidth=

Signature: void **box_dwidth=** (double w)

Description: Sets the width of the box in micrometer units

Applies to boxes only. Changes the width of the box to the value given in micrometer units and throws an exception if the shape is not a box. Translation to database units happens internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'box_dwidth'. This is the setter.

box_height

Signature: *[const]* int **box_height**

Description: Returns the height of the box

Applies to boxes only. Returns the height of the box and throws an exception if the shape is not a box.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'box_height'. This is the getter.

box_height=

Signature: void **box_height=** (int h)

Description: Sets the height of the box

Applies to boxes only. Changes the height of the box and throws an exception if the shape is not a box.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'box_height'. This is the setter.

box_p1

Signature: *[const]* Point **box_p1**

Description: Returns the lower left point of the box

Applies to boxes only. Returns the lower left point of the box and throws an exception if the shape is not a box.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'box_p1'. This is the getter.

box_p1=

(1) Signature: void **box_p1=** (const Point p)

Description: Sets the lower left point of the box

Applies to boxes only. Changes the lower left point of the box and throws an exception if the shape is not a box.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'box_p1'. This is the setter.

(2) Signature: void **box_p1=** (const DPoint p)

Description: Sets the lower left corner of the box with the point being given in micrometer units

Applies to boxes only. Changes the lower left point of the box and throws an exception if the shape is not a box. Translation from micrometer units to database units is done internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'box_p1'. This is the setter.
The object exposes a writable attribute 'box_dp1'. This is the setter.

box_p2

Signature: *[const]* Point **box_p2**

Description: Returns the upper right point of the box

Applies to boxes only. Returns the upper right point of the box and throws an exception if the shape is not a box.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'box_p2'. This is the getter.

box_p2=

(1) Signature: void **box_p2=** (const Point p)

Description: Sets the upper right point of the box

Applies to boxes only. Changes the upper right point of the box and throws an exception if the shape is not a box.

This method has been introduced in version 0.23.

Python specific notes:

The object exposes a writable attribute 'box_p2'. This is the setter.

(2) Signature: void **box_p2=** (const DPoint p)

Description: Sets the upper right corner of the box with the point being given in micrometer units

Applies to boxes only. Changes the upper right point of the box and throws an exception if the shape is not a box. Translation from micrometer units to database units is done internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'box_p2'. This is the setter.
The object exposes a writable attribute 'box_dp2'. This is the setter.

box_width

Signature: *[const]* int **box_width**

Description: Returns the width of the box

Applies to boxes only. Returns the width of the box and throws an exception if the shape is not a box.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'box_width'. This is the getter.

box_width=

Signature: void **box_width=** (int w)

Description: Sets the width of the box

Applies to boxes only. Changes the width of the box and throws an exception if the shape is not a box.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'box_width'. This is the setter.

cell

Signature: Cell ptr **cell**

Description: Gets a reference to the cell the shape belongs to

This reference can be nil, if the Shape object is not living inside a cell

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'cell'. This is the getter.

cell=

Signature: void **cell=** (Cell ptr cell)

Description: Moves the shape to a different cell

Both the current and the target cell must reside in the same layout.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'cell'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

darea

Signature: *[const]* double **darea**

Description: Returns the area of the shape in square micrometer units

This method has been added in version 0.25.

dbbox

Signature: *[const]* DBox **dbbox**

Description: Returns the bounding box of the shape in micrometer units

This method has been added in version 0.25.

dbox

Signature: *[const]* variant **dbox**

Description: Gets the box object in micrometer units

See box for a description of this method. This method returns the box after translation to micrometer units.

This method has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'dbox'. This is the getter.

dbox=

Signature: void **dbox=** (const DBox box)

Description: Replaces the shape by the given box (in micrometer units)

This method replaces the shape by the given box, like box= with a Box argument does. This version translates the box from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'box'. This is the setter.
The object exposes a writable attribute 'dbox'. This is the setter.

dedge

Signature: *[const]* variant **dedge**

Description: Returns the edge object as a DEdge object in micrometer units

See edge for a description of this method. This method returns the edge after translation to micrometer units.

This method has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'dedge'. This is the getter.

dedge=

Signature: void **dedge=** (const DEdge edge)

Description: Replaces the shape by the given edge (in micrometer units)

This method replaces the shape by the given edge, like edge= with a Edge argument does. This version translates the edge from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'edge'. This is the setter.
The object exposes a writable attribute 'dedge'. This is the setter.

dedge_pair

Signature: *[const]* variant **dedge_pair**

Description: Returns the edge pair object as a DEdgePair object in micrometer units

See edge_pair for a description of this method. This method returns the edge pair after translation to micrometer units.

This method has been added in version 0.26.

Python specific notes:
The object exposes a readable attribute 'dedge_pair'. This is the getter.

dedge_pair=

Signature: void **dedge_pair=** (const DEdgePair edge_pair)

Description: Replaces the shape by the given edge pair (in micrometer units)

This method replaces the shape by the given edge pair, like edge_pair= with a EdgePair argument does. This version translates the edge pair from micrometer units to database units internally.

This method has been introduced in version 0.26.

Python specific notes:
The object exposes a writable attribute 'edge_pair'. This is the setter.
The object exposes a writable attribute 'dedge_pair'. This is the setter.

delete

Signature: void **delete**

Description: Deletes the shape

After the shape is deleted, the shape object is emptied and points to nothing.

This method has been introduced in version 0.23.

delete_property

Signature: void **delete_property** (variant key)

Description: Deletes the user property with the given key

This method is a convenience method that deletes the property with the given key. It does nothing if no property with that key exists. Using that method is more convenient than creating a new property set with a new ID and assigning that properties ID. This method may change the properties ID. Calling this method will invalidate any iterators. It should not be called inside a loop iterating over shapes.

This method has been introduced in version 0.22.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dpath

Signature: *[const]* variant **dpath**

Description: Returns the path object as a DPath object in micrometer units

See path for a description of this method. This method returns the path after translation to micrometer units.

This method has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'dpath'. This is the getter.

dpath=

Signature: void **dpath=** (const DPath path)

Description: Replaces the shape by the given path (in micrometer units)

This method replaces the shape by the given path, like path= with a Path argument does. This version translates the path from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'path'. This is the setter.
The object exposes a writable attribute 'dpath'. This is the setter.

dperimeter

Signature: *[const]* double **dperimeter**

Description: Returns the perimeter of the shape in micrometer units

This method will return an approximation of the perimeter for paths.

This method has been added in version 0.25.

dpoint

Signature: *[const]* variant **dpoint**

Description: Returns the point object as a DPoint object in micrometer units

See point for a description of this method. This method returns the point after translation to micrometer units.

This method has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'dpoint'. This is the getter.

dpoint=

Signature: void **dpoint=** (const DPoint point)

Description: Replaces the shape by the given point (in micrometer units)

This method replaces the shape by the given point, like point= with a Point argument does. This version translates the point from micrometer units to database units internally.

This method has been introduced in version 0.28.

Python specific notes:
The object exposes a writable attribute 'point'. This is the setter.
The object exposes a writable attribute 'dpoint'. This is the setter.

dpolygon

Signature: *[const]* variant **dpolygon**

Description: Returns the polygon object in micrometer units

Returns the polygon object that this shape refers to or converts the object to a polygon. The method returns the same object than polygon, but translates it to micrometer units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'dpolygon'. This is the getter.

dpolygon=

Signature: void **dpolygon=** (const DPolygon polygon)

Description: Replaces the shape by the given polygon (in micrometer units)

This method replaces the shape by the given polygon, like polygon= with a Polygon argument does. This version translates the polygon from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'polygon'. This is the setter.
The object exposes a writable attribute 'dpolygon'. This is the setter.

drectangle

Signature: *[const]* variant **drectangle**

Description: Gets the rectangle in micron units if the object represents one or nil if not

If the shape represents a rectangle - i.e. a box or box polygon, a path with two points and no round ends - this method returns the box. If not, nil is returned.

This method has been introduced in version 0.29.

dsimple_polygon

Signature: *[const]* variant **dsimple_polygon**

Description: Returns the simple polygon object in micrometer units

Returns the simple polygon object that this shape refers to or converts the object to a simple polygon. The method returns the same object than simple_polygon, but translates it to micrometer units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'dsimple_polygon'. This is the getter.

dsimple_polygon=

Signature: void **dsimple_polygon=** (const DSimplePolygon polygon)

Description: Replaces the shape by the given simple polygon (in micrometer units)

This method replaces the shape by the given text, like simple_polygon= with a SimplePolygon argument does. This version translates the polygon from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'simple_polygon'. This is the setter.
The object exposes a writable attribute 'dsimple_polygon'. This is the setter.

dtext

Signature: *[const]* variant **dtext**

Description: Returns the path object as a DText object in micrometer units

See text for a description of this method. This method returns the text after translation to micrometer units.

This method has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'dtext'. This is the getter.

dtext=

Signature: void **dtext=** (const DText text)

Description: Replaces the shape by the given text (in micrometer units)

This method replaces the shape by the given text, like text= with a Text argument does. This version translates the text from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'text'. This is the setter.

	The object exposes a writable attribute 'dtext'. This is the setter.
dup	**Signature**: *[const]* new Shape ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
each_dedge	**(1) Signature**: *[const,iter]* DEdge **each_dedge** **Description**: Iterates over the edges of the object and returns edges in micrometer units This method iterates over all edges of polygons and simple polygons like each_edge, but will deliver edges in micrometer units. Multiplication by the database unit is done internally. This method has been introduced in version 0.25. **(2) Signature**: *[const,iter]* DEdge **each_dedge** (unsigned int contour) **Description**: Iterates over the edges of a single contour of the object and returns edges in micrometer units This method iterates over all edges of polygons and simple polygons like each_edge, but will deliver edges in micrometer units. Multiplication by the database unit is done internally. This method has been introduced in version 0.25.
each_dpoint	**Signature**: *[const,iter]* DPoint **each_dpoint** **Description**: Iterates over all points of the object and returns points in micrometer units This method iterates over all points of the object like each_point, but it returns DPoint objects that are given in micrometer units already. Multiplication with the database unit happens internally. This method has been introduced in version 0.25.
each_dpoint_hole	**Signature**: *[const,iter]* DPoint **each_dpoint_hole** (unsigned int hole_index) **Description**: Iterates over a hole contour of the object and returns points in micrometer units This method iterates over all points of the object's contour' like each_point_hole, but it returns DPoint objects that are given in micrometer units already. Multiplication with the database unit happens internally. This method has been introduced in version 0.25.
each_dpoint_hull	**Signature**: *[const,iter]* DPoint **each_dpoint_hull** **Description**: Iterates over the hull contour of the object and returns points in micrometer units This method iterates over all points of the object's contour' like each_point_hull, but it returns DPoint objects that are given in micrometer units already. Multiplication with the database unit happens internally. This method has been introduced in version 0.25.
each_edge	**(1) Signature**: *[const,iter]* Edge **each_edge** **Description**: Iterates over the edges of the object This method applies to polygons and simple polygons and delivers all edges that form the polygon's contours. Hole edges are oriented counterclockwise while hull edges are oriented clockwise.

It will throw an exception if the object is not a polygon.

(2) Signature: *[const,iter]* Edge **each_edge** (unsigned int contour)

Description: Iterates over the edges of a single contour of the object

contour: The contour number (0 for hull, 1 for first hole ...)

This method applies to polygons and simple polygons and delivers all edges that form the given contour of the polygon. The hull has contour number 0, the first hole has contour 1 etc. Hole edges are oriented counterclockwise while hull edges are oriented clockwise.

It will throw an exception if the object is not a polygon.

This method was introduced in version 0.24.

each_point

Signature: *[const,iter]* Point **each_point**

Description: Iterates over all points of the object

This method applies to paths and delivers all points of the path's center line. It will throw an exception for other objects.

each_point_hole

Signature: *[const,iter]* Point **each_point_hole** (unsigned int hole_index)

Description: Iterates over the points of a hole contour

hole: The hole index (see holes () method)

This method applies to polygons and delivers all points of the respective hole contour. It will throw an exception for other objects. Simple polygons deliver an empty sequence.

each_point_hull

Signature: *[const,iter]* Point **each_point_hull**

Description: Iterates over the hull contour of the object

This method applies to polygons and delivers all points of the polygon hull contour. It will throw an exception for other objects.

edge

Signature: *[const]* variant **edge**

Description: Returns the edge object

Starting with version 0.23, this method returns nil, if the shape does not represent an edge.

Python specific notes:
The object exposes a readable attribute 'edge'. This is the getter.

edge=

(1) Signature: void **edge=** (const Edge edge)

Description: Replaces the shape by the given edge

This method replaces the shape by the given edge. This method can only be called for editable layouts. It does not change the user properties of the shape. Calling this method will invalidate any iterators. It should not be called inside a loop iterating over shapes.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'edge'. This is the setter.

(2) Signature: void **edge=** (const DEdge edge)

Description: Replaces the shape by the given edge (in micrometer units)

This method replaces the shape by the given edge, like edge= with a Edge argument does. This version translates the edge from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'edge'. This is the setter.
The object exposes a writable attribute 'dedge'. This is the setter.

edge_pair

Signature: *[const]* variant **edge_pair**

Description: Returns the edge pair object

This method has been introduced in version 0.26.

Python specific notes:
The object exposes a readable attribute 'edge_pair'. This is the getter.

edge_pair=

(1) Signature: void **edge_pair=** (const EdgePair edge_pair)

Description: Replaces the shape by the given edge pair

This method replaces the shape by the given edge pair. This method can only be called for editable layouts. It does not change the user properties of the shape. Calling this method will invalidate any iterators. It should not be called inside a loop iterating over shapes.

This method has been introduced in version 0.26.

Python specific notes:
The object exposes a writable attribute 'edge_pair'. This is the setter.

(2) Signature: void **edge_pair=** (const DEdgePair edge_pair)

Description: Replaces the shape by the given edge pair (in micrometer units)

This method replaces the shape by the given edge pair, like edge_pair= with a EdgePair argument does. This version translates the edge pair from micrometer units to database units internally.

This method has been introduced in version 0.26.

Python specific notes:
The object exposes a writable attribute 'edge_pair'. This is the setter.
The object exposes a writable attribute 'dedge_pair'. This is the setter.

has_prop_id?

Signature: *[const]* bool **has_prop_id?**

Description: Returns true, if the shape has properties, i.e. has a properties ID

holes

Signature: *[const]* unsigned int **holes**

Description: Returns the number of holes

This method applies to polygons and will throw an exception for other objects.. Simple polygons deliver a value of zero.

is_array_member?

Signature: *[const]* bool **is_array_member?**

Description: Returns true, if the shape is a member of a shape array

is_box?

Signature: *[const]* bool **is_box?**

Description: Returns true if the shape is a box

is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
is_edge?	**Signature**: *[const]* bool **is_edge?** **Description**: Returns true, if the object is an edge
is_edge_pair?	**Signature**: *[const]* bool **is_edge_pair?** **Description**: Returns true, if the object is an edge pair This method has been introduced in version 0.26.
is_null?	**Signature**: *[const]* bool **is_null?** **Description**: Returns true, if the shape reference is a null reference (not referring to a shape)
is_path?	**Signature**: *[const]* bool **is_path?** **Description**: Returns true, if the shape is a path
is_point?	**Signature**: *[const]* bool **is_point?** **Description**: Returns true, if the object is an point This method has been introduced in version 0.28.
is_polygon?	**Signature**: *[const]* bool **is_polygon?** **Description**: Returns true, if the shape is a polygon This method returns true only if the object is a polygon or a simple polygon. Other objects can convert to polygons, for example paths, so it may be possible to use the polygon method also if is_polygon? does not return true.
is_simple_polygon?	**Signature**: *[const]* bool **is_simple_polygon?** **Description**: Returns true, if the shape is a simple polygon This method returns true only if the object is a simple polygon. The simple polygon identity is contained in the polygon identity, so usually it is sufficient to use is_polygon? and polygon instead of specifically handle simply polygons. This method is provided only for specific optimisation purposes.
is_text?	**Signature**: *[const]* bool **is_text?** **Description**: Returns true, if the object is a text
is_user_object?	**Signature**: *[const]* bool **is_user_object?** **Description**: Returns true if the shape is a user defined object
is_valid?	**Signature**: *[const]* bool **is_valid?** **Description**: Returns true, if the shape is valid

After the shape is deleted, the shape object is no longer valid and this method returns false.

This method has been introduced in version 0.23.

layer

Signature: *[const]* unsigned int **layer**

Description: Returns the layer index of the layer the shape is on

Throws an exception if the shape does not reside inside a cell.

This method has been added in version 0.23.

Python specific notes:
The object exposes a readable attribute 'layer'. This is the getter.

layer=

Signature: void **layer=** (unsigned int layer_index)

Description: Moves the shape to a layer given by the layer index object

This method has been added in version 0.23.

Python specific notes:
The object exposes a writable attribute 'layer'. This is the setter.

layer_info

Signature: *[const]* LayerInfo **layer_info**

Description: Returns the LayerInfo object of the layer the shape is on

If the shape does not reside inside a cell, an empty layer is returned.

This method has been added in version 0.23.

Python specific notes:
The object exposes a readable attribute 'layer_info'. This is the getter.

layer_info=

Signature: void **layer_info=** (const LayerInfo layer_info)

Description: Moves the shape to a layer given by a LayerInfo object

If no layer with the given properties exists, an exception is thrown.

This method has been added in version 0.23.

Python specific notes:
The object exposes a writable attribute 'layer_info'. This is the setter.

layout

Signature: Layout ptr **layout**

Description: Gets a reference to the Layout the shape belongs to

This reference can be nil, if the Shape object is not living inside a layout.

This method has been introduced in version 0.22.

new

Signature: *[static]* new Shape ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

path

Signature: *[const]* variant **path**

Description: Returns the path object

Starting with version 0.23, this method returns nil, if the shape does not represent a path.

Python specific notes:

The object exposes a readable attribute 'path'. This is the getter.

path=

(1) Signature: void **path=** (const Path box)

Description: Replaces the shape by the given path object

This method replaces the shape by the given path object. This method can only be called for editable layouts. It does not change the user properties of the shape. Calling this method will invalidate any iterators. It should not be called inside a loop iterating over shapes.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'path'. This is the setter.

(2) Signature: void **path=** (const DPath path)

Description: Replaces the shape by the given path (in micrometer units)

This method replaces the shape by the given path, like path= with a Path argument does. This version translates the path from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'path'. This is the setter.
The object exposes a writable attribute 'dpath'. This is the setter.

path_bgnext

Signature: *[const]* int **path_bgnext**

Description: Gets the path's starting vertex extension

Applies to paths only. Will throw an exception if the object is not a path.

Python specific notes:
The object exposes a readable attribute 'path_bgnext'. This is the getter.

path_bgnext=

Signature: void **path_bgnext=** (int e)

Description: Sets the path's starting vertex extension

Applies to paths only. Will throw an exception if the object is not a path.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'path_bgnext'. This is the setter.

path_dbgnext

Signature: *[const]* double **path_dbgnext**

Description: Gets the path's starting vertex extension in micrometer units

Applies to paths only. Will throw an exception if the object is not a path.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'path_dbgnext'. This is the getter.

path_dbgnext=

Signature: void **path_dbgnext=** (double e)

Description: Sets the path's starting vertex extension in micrometer units

Applies to paths only. Will throw an exception if the object is not a path.

This method has been introduced in version 0.25.

Python specific notes:

	The object exposes a writable attribute 'path_dbgnext'. This is the setter.
path_dendext	**Signature**: *[const]* double **path_dendext** **Description**: Gets the path's end vertex extension in micrometer units Applies to paths only. Will throw an exception if the object is not a path. This method has been introduced in version 0.25. **Python specific notes:** The object exposes a readable attribute 'path_dendext'. This is the getter.
path_dendext=	**Signature**: void **path_dendext=** (double e) **Description**: Sets the path's end vertex extension in micrometer units Applies to paths only. Will throw an exception if the object is not a path. This method has been introduced in version 0.25. **Python specific notes:** The object exposes a writable attribute 'path_dendext'. This is the setter.
path_dlength	**Signature**: *[const]* double **path_dlength** **Description**: Returns the length of the path in micrometer units Applies to paths only. Will throw an exception if the object is not a path. This method returns the length of the spine plus extensions if present. The value returned is given in micrometer units. This method has been added in version 0.25.
path_dwidth	**Signature**: *[const]* double **path_dwidth** **Description**: Gets the path width in micrometer units Applies to paths only. Will throw an exception if the object is not a path. This method has been introduced in version 0.25. **Python specific notes:** The object exposes a readable attribute 'path_dwidth'. This is the getter.
path_dwidth=	**Signature**: void **path_dwidth=** (double w) **Description**: Sets the path width in micrometer units Applies to paths only. Will throw an exception if the object is not a path. Conversion to database units is done internally. This method has been introduced in version 0.25. **Python specific notes:** The object exposes a writable attribute 'path_dwidth'. This is the setter.
path_endext	**Signature**: *[const]* int **path_endext** **Description**: Obtain the path's end vertex extension Applies to paths only. Will throw an exception if the object is not a path. **Python specific notes:** The object exposes a readable attribute 'path_endext'. This is the getter.
path_endext=	**Signature**: void **path_endext=** (int e)

Description: Sets the path's end vertex extension

Applies to paths only. Will throw an exception if the object is not a path.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'path_endext'. This is the setter.

path_length

Signature: *[const]* int **path_length**

Description: Returns the length of the path

Applies to paths only. Will throw an exception if the object is not a path. This method returns the length of the spine plus extensions if present.

This method has been added in version 0.23.

path_width

Signature: *[const]* int **path_width**

Description: Gets the path width

Applies to paths only. Will throw an exception if the object is not a path.

Python specific notes:
The object exposes a readable attribute 'path_width'. This is the getter.

path_width=

Signature: void **path_width=** (int w)

Description: Sets the path width

Applies to paths only. Will throw an exception if the object is not a path.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'path_width'. This is the setter.

perimeter

Signature: *[const]* unsigned long **perimeter**

Description: Returns the perimeter of the shape

This method will return an approximation of the perimeter for paths.

This method has been added in version 0.23.

point

Signature: *[const]* variant **point**

Description: Returns the point object

This method has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'point'. This is the getter.

point=

(1) Signature: void **point=** (const Point point)

Description: Replaces the shape by the given point

This method replaces the shape by the given point. This method can only be called for editable layouts. It does not change the user properties of the shape. Calling this method will invalidate any iterators. It should not be called inside a loop iterating over shapes.

This method has been introduced in version 0.28.

Python specific notes:
The object exposes a writable attribute 'point'. This is the setter.

(2) Signature: void **point=** (const DPoint point)

Description: Replaces the shape by the given point (in micrometer units)

This method replaces the shape by the given point, like point= with a Point argument does. This version translates the point from micrometer units to database units internally.

This method has been introduced in version 0.28.

Python specific notes:
The object exposes a writable attribute 'point'. This is the setter.
The object exposes a writable attribute 'dpoint'. This is the setter.

polygon

Signature: *[const]* variant **polygon**

Description: Returns the polygon object

Returns the polygon object that this shape refers to or converts the object to a polygon. Paths, boxes and simple polygons are converted to polygons. For paths this operation renders the path's hull contour.

Starting with version 0.23, this method returns nil, if the shape does not represent a geometrical primitive that can be converted to a polygon.

Python specific notes:
The object exposes a readable attribute 'polygon'. This is the getter.

polygon=

(1) Signature: void **polygon=** (const Polygon box)

Description: Replaces the shape by the given polygon object

This method replaces the shape by the given polygon object. This method can only be called for editable layouts. It does not change the user properties of the shape. Calling this method will invalidate any iterators. It should not be called inside a loop iterating over shapes.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'polygon'. This is the setter.

(2) Signature: void **polygon=** (const DPolygon polygon)

Description: Replaces the shape by the given polygon (in micrometer units)

This method replaces the shape by the given polygon, like polygon= with a Polygon argument does. This version translates the polygon from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'polygon'. This is the setter.
The object exposes a writable attribute 'dpolygon'. This is the setter.

prop_id

Signature: *[const]* unsigned long **prop_id**

Description: Gets the properties ID associated with the shape

The Layout object can be used to retrieve the actual properties associated with the ID.

Python specific notes:
The object exposes a readable attribute 'prop_id'. This is the getter.

prop_id=

Signature: void **prop_id=** (unsigned long id)

Description: Sets the properties ID of this shape

The Layout object can be used to retrieve an ID for a given set of properties. Calling this method will invalidate any iterators. It should not be called inside a loop iterating over shapes.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'prop_id'. This is the setter.

property

Signature: *[const]* variant **property** (variant key)

Description: Gets the user property with the given key

This method is a convenience method that gets the property with the given key. If no property with that key does not exist, it will return nil. Using that method is more convenient than using the layout object and the properties ID to retrieve the property value. This method has been introduced in version 0.22.

rectangle

Signature: *[const]* variant **rectangle**

Description: Gets the rectangle if the object represents one or nil if not

If the shape represents a rectangle - i.e. a box or box polygon, a path with two points and no round ends - this method returns the box. If not, nil is returned.

This method has been introduced in version 0.29.

round_path=

Signature: void **round_path=** (bool r)

Description: The path will be a round-ended path if this property is set to true

Applies to paths only. Will throw an exception if the object is not a path. Please note that the extensions will apply as well. To get a path with circular ends, set the begin and end extensions to half the path's width.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'round_path'. This is the setter.

round_path?

Signature: *[const]* bool **round_path?**

Description: Returns true, if the path has round ends

Applies to paths only. Will throw an exception if the object is not a path.

Python specific notes:
The object exposes a readable attribute 'round_path'. This is the getter.

set_property

Signature: void **set_property** (variant key, variant value)

Description: Sets the user property with the given key to the given value

This method is a convenience method that sets the property with the given key to the given value. If no property with that key exists, it will create one. Using that method is more convenient than creating a new property set with a new ID and assigning that properties ID. This method may change the properties ID. Note: GDS only supports integer keys. OASIS supports numeric and string keys. Calling this method will invalidate any iterators. It should not be called inside a loop iterating over shapes.

This method has been introduced in version 0.22.

shapes

Signature: Shapes ptr **shapes**

Description: Gets a reference to the Shapes container the shape lives in

This reference can be nil, if the Shape object is not referring to an actual shape.

This method has been introduced in version 0.22.

simple_polygon

Signature: *[const]* variant **simple_polygon**

Description: Returns the simple polygon object

Returns the simple polygon object that this shape refers to or converts the object to a simple polygon. Paths, boxes and polygons are converted to simple polygons. Polygons with holes will have their holes removed but introducing cut lines that connect the hole contours with the outer contour. Starting with version 0.23, this method returns nil, if the shape does not represent a geometrical primitive that can be converted to a simple polygon.

Python specific notes:
The object exposes a readable attribute 'simple_polygon'. This is the getter.

simple_polygon=

(1) Signature: void **simple_polygon=** (const SimplePolygon polygon)

Description: Replaces the shape by the given simple polygon object

This method replaces the shape by the given simple polygon object. This method can only be called for editable layouts. It does not change the user properties of the shape. Calling this method will invalidate any iterators. It should not be called inside a loop iterating over shapes.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'simple_polygon'. This is the setter.

(2) Signature: void **simple_polygon=** (const DSimplePolygon polygon)

Description: Replaces the shape by the given simple polygon (in micrometer units)

This method replaces the shape by the given text, like simple_polygon= with a SimplePolygon argument does. This version translates the polygon from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'simple_polygon'. This is the setter.
The object exposes a writable attribute 'dsimple_polygon'. This is the setter.

t_box

Signature: *[static]* int **t_box**

Description:

Use of this method is deprecated. Use TBox instead

Python specific notes:
The object exposes a readable attribute 'TBox'. This is the getter.

t_box_array

Signature: *[static]* int **t_box_array**

Description:

Use of this method is deprecated. Use TBoxArray instead

Python specific notes:
The object exposes a readable attribute 'TBoxArray'. This is the getter.

t_box_array_member

Signature: *[static]* int **t_box_array_member**

Description:

Use of this method is deprecated. Use TBoxArrayMember instead

Python specific notes:
The object exposes a readable attribute 'TBoxArrayMember'. This is the getter.

t_edge

Signature: *[static]* int **t_edge**

Description:

Use of this method is deprecated. Use TEdge instead

Python specific notes:
The object exposes a readable attribute 'TEdge'. This is the getter.

t_edge_pair

Signature: *[static]* int **t_edge_pair**

Description:

Use of this method is deprecated. Use TEdgePair instead

Python specific notes:
The object exposes a readable attribute 'TEdgePair'. This is the getter.

t_null

Signature: *[static]* int **t_null**

Description:

Use of this method is deprecated. Use TNull instead

Python specific notes:
The object exposes a readable attribute 'TNull'. This is the getter.

t_path

Signature: *[static]* int **t_path**

Description:

Use of this method is deprecated. Use TPath instead

Python specific notes:
The object exposes a readable attribute 'TPath'. This is the getter.

t_path_ptr_array

Signature: *[static]* int **t_path_ptr_array**

Description:

Use of this method is deprecated. Use TPathPtrArray instead

Python specific notes:
The object exposes a readable attribute 'TPathPtrArray'. This is the getter.

t_path_ptr_array_member

Signature: *[static]* int **t_path_ptr_array_member**

Description:

Use of this method is deprecated. Use TPathPtrArrayMember instead

Python specific notes:
The object exposes a readable attribute 'TPathPtrArrayMember'. This is the getter.

t_path_ref

Signature: *[static]* int **t_path_ref**

Description:

Use of this method is deprecated. Use TPathRef instead

Python specific notes:
The object exposes a readable attribute 'TPathRef'. This is the getter.

t_point

Signature: *[static]* int **t_point**

Description:

Use of this method is deprecated. Use TPoint instead

Python specific notes:
The object exposes a readable attribute 'TPoint'. This is the getter.

t_polygon

Signature: *[static]* int **t_polygon**

Description:

Use of this method is deprecated. Use TPolygon instead

Python specific notes:
The object exposes a readable attribute 'TPolygon'. This is the getter.

t_polygon_ptr_array

Signature: *[static]* int **t_polygon_ptr_array**

Description:

Use of this method is deprecated. Use TPolygonPtrArray instead

Python specific notes:
The object exposes a readable attribute 'TPolygonPtrArray'. This is the getter.

t_polygon_ptr_array_member

Signature: *[static]* int **t_polygon_ptr_array_member**

Description:

Use of this method is deprecated. Use TPolygonPtrArrayMember instead

Python specific notes:
The object exposes a readable attribute 'TPolygonPtrArrayMember'. This is the getter.

t_polygon_ref

Signature: *[static]* int **t_polygon_ref**

Description:

Use of this method is deprecated. Use TPolygonRef instead

Python specific notes:
The object exposes a readable attribute 'TPolygonRef'. This is the getter.

t_short_box

Signature: *[static]* int **t_short_box**

Description:

Use of this method is deprecated. Use TShortBox instead

Python specific notes:
The object exposes a readable attribute 'TShortBox'. This is the getter.

t_short_box_array

Signature: *[static]* int **t_short_box_array**

Description:

Use of this method is deprecated. Use TShortBoxArray instead

Python specific notes:
The object exposes a readable attribute 'TShortBoxArray'. This is the getter.

t_short_box_array_member

Signature: *[static]* int **t_short_box_array_member**

Description:

Use of this method is deprecated. Use TShortBoxArrayMember instead

Python specific notes:
The object exposes a readable attribute 'TShortBoxArrayMember'. This is the getter.

t_simple_polygon

Signature: *[static]* int **t_simple_polygon**

Description:

Use of this method is deprecated. Use TSimplePolygon instead

Python specific notes:
The object exposes a readable attribute 'TSimplePolygon'. This is the getter.

t_simple_polygon_ptr_array

Signature: *[static]* int **t_simple_polygon_ptr_array**

Description:

Use of this method is deprecated. Use TSimplePolygonPtrArray instead

Python specific notes:
The object exposes a readable attribute 'TSimplePolygonPtrArray'. This is the getter.

t_simple_polygon_ptr_array

Signature: *[static]* int **t_simple_polygon_ptr_array_member**

Description:

Use of this method is deprecated. Use TSimplePolygonPtrArrayMember instead

Python specific notes:
The object exposes a readable attribute 'TSimplePolygonPtrArrayMember'. This is the getter.

t_simple_polygon_ref

Signature: *[static]* int **t_simple_polygon_ref**

Description:

Use of this method is deprecated. Use TSimplePolygonRef instead

Python specific notes:
The object exposes a readable attribute 'TSimplePolygonRef'. This is the getter.

t_text

Signature: *[static]* int **t_text**

Description:

Use of this method is deprecated. Use TText instead

Python specific notes:
The object exposes a readable attribute 'TText'. This is the getter.

t_text_ptr_array

Signature: *[static]* int **t_text_ptr_array**

Description:

Use of this method is deprecated. Use TTextPtrArray instead

Python specific notes:
The object exposes a readable attribute 'TTextPtrArray'. This is the getter.

t_text_ptr_array_member

Signature: *[static]* int **t_text_ptr_array_member**

Description:

Use of this method is deprecated. Use TTextPtrArrayMember instead

Python specific notes:
The object exposes a readable attribute 'TTextPtrArrayMember'. This is the getter.

t_text_ref

Signature: *[static]* int **t_text_ref**

Description:

Use of this method is deprecated. Use TTextRef instead

Python specific notes:
The object exposes a readable attribute 'TTextRef'. This is the getter.

t_user_object

Signature: *[static]* int **t_user_object**

Description:

Use of this method is deprecated. Use TUserObject instead

Python specific notes:
The object exposes a readable attribute 'TUserObject'. This is the getter.

text

Signature: *[const]* variant **text**

Description: Returns the text object

Starting with version 0.23, this method returns nil, if the shape does not represent a text.

Python specific notes:
The object exposes a readable attribute 'text'. This is the getter.

text=

(1) Signature: void **text=** (const Text box)

Description: Replaces the shape by the given text object

This method replaces the shape by the given text object. This method can only be called for editable layouts. It does not change the user properties of the shape. Calling this method will invalidate any iterators. It should not be called inside a loop iterating over shapes.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'text'. This is the setter.

(2) Signature: void **text=** (const DText text)

Description: Replaces the shape by the given text (in micrometer units)

This method replaces the shape by the given text, like text= with a Text argument does. This version translates the text from micrometer units to database units internally.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'text'. This is the setter.
The object exposes a writable attribute 'dtext'. This is the setter.

text_dpos

Signature: *[const]* DVector **text_dpos**

Description: Gets the text's position in micrometer units

Applies to texts only. Will throw an exception if the object is not a text.

This method has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'text_dpos'. This is the getter.

text_dpos=

Signature: void **text_dpos=** (const DVector p)

Description: Sets the text's position in micrometer units

Applies to texts only. Will throw an exception if the object is not a text.

This method has been added in version 0.25.

Python specific notes:
The object exposes a writable attribute 'text_pos'. This is the setter.

	The object exposes a writable attribute 'text_dpos'. This is the setter.
text_dsize	**Signature**: *[const]* double **text_dsize** **Description**: Gets the text size in micrometer units Applies to texts only. Will throw an exception if the object is not a text. This method has been introduced in version 0.25. **Python specific notes:** The object exposes a readable attribute 'text_dsize'. This is the getter.
text_dsize=	**Signature**: void **text_dsize=** (double size) **Description**: Sets the text size in micrometer units Applies to texts only. Will throw an exception if the object is not a text. This method has been introduced in version 0.25. **Python specific notes:** The object exposes a writable attribute 'text_dsize'. This is the setter.
text_dtrans	**Signature**: *[const]* DTrans **text_dtrans** **Description**: Gets the text transformation in micrometer units Applies to texts only. Will throw an exception if the object is not a text. This method has been added in version 0.25. **Python specific notes:** The object exposes a readable attribute 'text_dtrans'. This is the getter.
text_dtrans=	**Signature**: void **text_dtrans=** (const DTrans trans) **Description**: Sets the text transformation in micrometer units Applies to texts only. Will throw an exception if the object is not a text. This method has been introduced in version 0.25. **Python specific notes:** The object exposes a writable attribute 'text_trans'. This is the setter. The object exposes a writable attribute 'text_dtrans'. This is the setter.
text_font	**Signature**: *[const]* int **text_font** **Description**: Gets the text's font Applies to texts only. Will throw an exception if the object is not a text. **Python specific notes:** The object exposes a readable attribute 'text_font'. This is the getter.
text_font=	**Signature**: void **text_font=** (int font) **Description**: Sets the text's font Applies to texts only. Will throw an exception if the object is not a text. This method has been introduced in version 0.23. **Python specific notes:** The object exposes a writable attribute 'text_font'. This is the setter.

text_halign

Signature: *[const]* int **text_halign**

Description: Gets the text's horizontal alignment

Applies to texts only. Will throw an exception if the object is not a text. The return value is 0 for left alignment, 1 for center alignment and 2 to right alignment.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'text_halign'. This is the getter.

text_halign=

Signature: void **text_halign=** (int a)

Description: Sets the text's horizontal alignment

Applies to texts only. Will throw an exception if the object is not a text. See text_halign for a description of that property.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'text_halign'. This is the setter.

text_pos

Signature: *[const]* Vector **text_pos**

Description: Gets the text's position

Applies to texts only. Will throw an exception if the object is not a text.

Python specific notes:
The object exposes a readable attribute 'text_pos'. This is the getter.

text_pos=

(1) Signature: void **text_pos=** (const Vector p)

Description: Sets the text's position

Applies to texts only. Will throw an exception if the object is not a text.

Python specific notes:
The object exposes a writable attribute 'text_pos'. This is the setter.

(2) Signature: void **text_pos=** (const DVector p)

Description: Sets the text's position in micrometer units

Applies to texts only. Will throw an exception if the object is not a text.

This method has been added in version 0.25.

Python specific notes:
The object exposes a writable attribute 'text_pos'. This is the setter.
The object exposes a writable attribute 'text_dpos'. This is the setter.

text_rot

Signature: *[const]* int **text_rot**

Description: Gets the text's orientation code (see Trans)

Applies to texts only. Will throw an exception if the object is not a text.

Python specific notes:
The object exposes a readable attribute 'text_rot'. This is the getter.

text_rot=

Signature: void **text_rot=** (int o)

Description: Sets the text's orientation code (see Trans)

Applies to texts only. Will throw an exception if the object is not a text.

Python specific notes:
The object exposes a writable attribute 'text_rot'. This is the setter.

text_size

Signature: *[const]* int **text_size**

Description: Gets the text size

Applies to texts only. Will throw an exception if the object is not a text.

Python specific notes:
The object exposes a readable attribute 'text_size'. This is the getter.

text_size=

Signature: void **text_size=** (int size)

Description: Sets the text size

Applies to texts only. Will throw an exception if the object is not a text.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'text_size'. This is the setter.

text_string

Signature: *[const]* string **text_string**

Description: Obtain the text string

Applies to texts only. Will throw an exception if the object is not a text.

Python specific notes:
The object exposes a readable attribute 'text_string'. This is the getter.

text_string=

Signature: void **text_string=** (string string)

Description: Sets the text string

Applies to texts only. Will throw an exception if the object is not a text.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'text_string'. This is the setter.

text_trans

Signature: *[const]* Trans **text_trans**

Description: Gets the text transformation

Applies to texts only. Will throw an exception if the object is not a text.

Python specific notes:
The object exposes a readable attribute 'text_trans'. This is the getter.

text_trans=

(1) Signature: void **text_trans=** (const Trans trans)

Description: Sets the text transformation

Applies to texts only. Will throw an exception if the object is not a text.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'text_trans'. This is the setter.

(2) Signature: void **text_trans=** (const DTrans trans)

Description: Sets the text transformation in micrometer units

Applies to texts only. Will throw an exception if the object is not a text.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'text_trans'. This is the setter.
The object exposes a writable attribute 'text_dtrans'. This is the setter.

text_valign

Signature: *[const]* int **text_valign**

Description: Gets the text's vertical alignment

Applies to texts only. Will throw an exception if the object is not a text. The return value is 0 for top alignment, 1 for center alignment and 2 to bottom alignment.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'text_valign'. This is the getter.

text_valign=

Signature: void **text_valign=** (int a)

Description: Sets the text's vertical alignment

Applies to texts only. Will throw an exception if the object is not a text. See text_valign for a description of that property.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'text_valign'. This is the setter.

to_s

Signature: *[const]* string **to_s**

Description: Create a string showing the contents of the reference

This method has been introduced with version 0.16.

Python specific notes:
This method is also available as 'str(object)'.

transform

(1) Signature: void **transform** (const Trans trans)

Description: Transforms the shape with the given transformation

This method has been introduced in version 0.23.

(2) Signature: void **transform** (const DTrans trans)

Description: Transforms the shape with the given transformation, given in micrometer units

This method has been introduced in version 0.25.

(3) Signature: void **transform** (const ICplxTrans trans)

Description: Transforms the shape with the given complex transformation

This method has been introduced in version 0.23.

(4) Signature: void **transform** (const DCplxTrans trans)

Description: Transforms the shape with the given complex transformation, given in micrometer units

This method has been introduced in version 0.25.

type

Signature: *[const]* int **type**

Description: Return the type of the shape

The returned values are the t_... constants available through the corresponding class members.

4.95. API reference - Class ShapeProcessor

Notation used in Ruby API documentation

Module: db

Description: The shape processor (boolean, sizing, merge on shapes)

The shape processor implements the boolean and edge set operations (size, merge). Because the shape processor might allocate resources which can be reused in later operations, it is implemented as an object that can be used several times. The shape processor is similar to the EdgeProcessor. The latter is specialized on handling polygons and edges directly.

Public constructors

new ShapeProcessor ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const ShapeProcessor other)	Assigns another object to self
	void	**boolean**	(const Layout layout_a, const Cell cell_a, unsigned int layer_a, const Layout layout_b, const Cell cell_b, unsigned int layer_b, Shapes out, int mode, bool hierarchical, bool resolve_holes, bool min_coherence)	Boolean operation on shapes from layouts
	Edge[]	**boolean**	(Shape[] in_a, CplxTrans[] trans_a, Shape[] in_b, CplxTrans[] trans_b, int mode)	Boolean operation on two given shape sets into an edge set
	Edge[]	**boolean**	(Shape[] in_a, Shape[] in_b, int mode)	Boolean operation on two given shape sets into an edge set

	Polygon[]	**boolean_to_polygo**	(Shape[] in_a, CplxTrans[] trans_a, Shape[] in_b, CplxTrans[] trans_b, int mode, bool resolve_holes, bool min_coherence)	Boolean operation on two given shape sets into a polygon set
	Polygon[]	**boolean_to_polygon**	(Shape[] in_a, Shape[] in_b, int mode, bool resolve_holes, bool min_coherence)	Boolean operation on two given shape sets into a polygon set
[const]	new ShapeProcessor ptr	**dup**		Creates a copy of self
	void	**merge**	(const Layout layout, const Cell cell, unsigned int layer, Shapes out, bool hierarchical, unsigned int min_wc, bool resolve_holes, bool min_coherence)	Merge the given shapes from a layout into a shapes container
	Edge[]	**merge**	(Shape[] in, CplxTrans[] trans, unsigned int min_wc)	Merge the given shapes
	Edge[]	**merge**	(Shape[] in, unsigned int min_wc)	Merge the given shapes
	Polygon[]	**merge_to_polygon**	(Shape[] in, CplxTrans[] trans, unsigned int min_wc, bool resolve_holes, bool min_coherence)	Merge the given shapes
	Polygon[]	**merge_to_polygon**	(Shape[] in, unsigned int min_wc, bool resolve_holes, bool min_coherence)	Merge the given shapes
	void	**size**	(const Layout layout, const Cell cell, unsigned int layer, Shapes out, int dx, int dy, unsigned int mode, bool hierarchical, bool resolve_holes, bool min_coherence)	Sizing operation on shapes from layouts
	void	**size**	(const Layout layout, const Cell cell, unsigned int layer, Shapes out,	Sizing operation on shapes from layouts

		int d, unsigned int mode, bool hierarchical, bool resolve_holes, bool min_coherence)	
Edge[]	size	(Shape[] in, CplxTrans[] trans, int d, unsigned int mode)	Size the given shapes
Edge[]	size	(Shape[] in, CplxTrans[] trans, int dx, int dy, unsigned int mode)	Size the given shapes
Edge[]	size	(Shape[] in, int d, unsigned int mode)	Size the given shapes
Edge[]	size	(Shape[] in, int dx, int dy, unsigned int mode)	Size the given shapes
Polygon[]	size_to_polygon	(Shape[] in, CplxTrans[] trans, int d, unsigned int mode, bool resolve_holes, bool min_coherence)	Size the given shapes
Polygon[]	size_to_polygon	(Shape[] in, CplxTrans[] trans, int dx, int dy, unsigned int mode, bool resolve_holes, bool min_coherence)	Size the given shapes
Polygon[]	size_to_polygon	(Shape[] in, int d, unsigned int mode, bool resolve_holes, bool min_coherence)	Size the given shapes
Polygon[]	size_to_polygon	(Shape[] in, int dx, int dy, unsigned int mode, bool resolve_holes, bool min_coherence)	Size the given shapes

Deprecated methods (protected, public, static, non-static and constructors)

void	create	Use of this method is deprecated. Use _create instead
void	destroy	Use of this method is deprecated. Use _destroy instead

[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const ShapeProcessor other)

Description: Assigns another object to self

boolean

(1) Signature: void **boolean** (const Layout layout_a, const Cell cell_a, unsigned int layer_a, const Layout layout_b, const Cell cell_b, unsigned int layer_b, Shapes out, int mode, bool hierarchical, bool resolve_holes, bool min_coherence)

Description: Boolean operation on shapes from layouts

layout_a:	The layout from which to take the shapes for input A
cell_a:	The cell (in 'layout') from which to take the shapes for input A
layer_a:	The cell (in 'layout') from which to take the shapes for input A
layout_b:	The layout from which to take the shapes for input B
cell_b:	The cell (in 'layout') from which to take the shapes for input B
layer_b:	The cell (in 'layout') from which to take the shapes for input B
out:	The shapes container where to put the shapes into (is cleared before)
mode:	The boolean operation (see EdgeProcessor)
hierarchical:	Collect shapes from sub cells as well
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the boolean operations. This implementation takes shapes from layout cells (optionally all in hierarchy) and produces new shapes in a shapes container.

(2) Signature: Edge[] **boolean** (Shape[] in_a, CplxTrans[] trans_a, Shape[] in_b, CplxTrans[] trans_b, int mode)

Description: Boolean operation on two given shape sets into an edge set

in_a:	The set of shapes to use for input A
trans_a:	A set of transformations to apply before the shapes are used
in_b:	The set of shapes to use for input A
trans_b:	A set of transformations to apply before the shapes are used
mode:	The boolean operation (see EdgeProcessor)

See the EdgeProcessor for a description of the boolean operations. This implementation takes shapes rather than polygons for input and produces an edge set.

(3) Signature: Edge[] **boolean** (Shape[] in_a, Shape[] in_b, int mode)

Description: Boolean operation on two given shape sets into an edge set

in_a:	The set of shapes to use for input A
in_b:	The set of shapes to use for input A
mode:	The boolean operation (see EdgeProcessor)

See the EdgeProcessor for a description of the boolean operations. This implementation takes shapes rather than polygons for input and produces an edge set.

This version does not feature a transformation for each shape (unity is assumed).

boolean_to_polygon

(1) Signature: Polygon[] **boolean_to_polygon** (Shape[] in_a, CplxTrans[] trans_a, Shape[] in_b, CplxTrans[] trans_b, int mode, bool resolve_holes, bool min_coherence)

Description: Boolean operation on two given shape sets into a polygon set

in_a:	The set of shapes to use for input A
trans_a:	A set of transformations to apply before the shapes are used
in_b:	The set of shapes to use for input A
trans_b:	A set of transformations to apply before the shapes are used
mode:	The boolean operation (see EdgeProcessor)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the boolean operations. This implementation takes shapes rather than polygons for input and produces a polygon set.

(2) Signature: Polygon[] **boolean_to_polygon** (Shape[] in_a, Shape[] in_b, int mode, bool resolve_holes, bool min_coherence)

Description: Boolean operation on two given shape sets into a polygon set

in_a:	The set of shapes to use for input A
in_b:	The set of shapes to use for input A
mode:	The boolean operation (see EdgeProcessor)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the boolean operations. This implementation takes shapes rather than polygons for input and produces a polygon set.

This version does not feature a transformation for each shape (unity is assumed).

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new ShapeProcessor ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

merge

(1) Signature: void **merge** (const Layout layout, const Cell cell, unsigned int layer, Shapes out, bool hierarchical, unsigned int min_wc, bool resolve_holes, bool min_coherence)

Description: Merge the given shapes from a layout into a shapes container

layout:	The layout from which to take the shapes
cell:	The cell (in 'layout') from which to take the shapes
layer:	The cell (in 'layout') from which to take the shapes
out:	The shapes container where to put the shapes into (is cleared before)
hierarchical:	Collect shapes from sub cells as well
min_wc:	The minimum wrap count for output (0: all polygons, 1: at least two overlapping)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the merge method. This implementation takes shapes from a layout cell (optionally all in hierarchy) and produces new shapes in a shapes container.

(2) Signature: Edge[] **merge** (Shape[] in, CplxTrans[] trans, unsigned int min_wc)

Description: Merge the given shapes

in:	The set of shapes to merge
trans:	A corresponding set of transformations to apply on the shapes
min_wc:	The minimum wrap count for output (0: all polygons, 1: at least two overlapping)

See the EdgeProcessor for a description of the merge method. This implementation takes shapes rather than polygons for input and produces an edge set.

(3) Signature: Edge[] **merge** (Shape[] in, unsigned int min_wc)

Description: Merge the given shapes

in:	The set of shapes to merge
min_wc:	The minimum wrap count for output (0: all polygons, 1: at least two overlapping)

See the EdgeProcessor for a description of the merge method. This implementation takes shapes rather than polygons for input and produces an edge set.

This version does not feature a transformation for each shape (unity is assumed).

merge_to_polygon

(1) Signature: Polygon[] **merge_to_polygon** (Shape[] in, CplxTrans[] trans, unsigned int min_wc, bool resolve_holes, bool min_coherence)

Description: Merge the given shapes

in:	The set of shapes to merge
trans:	A corresponding set of transformations to apply on the shapes
min_wc:	The minimum wrap count for output (0: all polygons, 1: at least two overlapping)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the merge method. This implementation takes shapes rather than polygons for input and produces a polygon set.

(2) Signature: Polygon[] **merge_to_polygon** (Shape[] in, unsigned int min_wc, bool resolve_holes, bool min_coherence)

Description: Merge the given shapes

in:	The set of shapes to merge
min_wc:	The minimum wrap count for output (0: all polygons, 1: at least two overlapping)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the merge method. This implementation takes shapes rather than polygons for input and produces a polygon set.

This version does not feature a transformation for each shape (unity is assumed).

new

Signature: *[static]* new ShapeProcessor ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

size

(1) Signature: void **size** (const Layout layout, const Cell cell, unsigned int layer, Shapes out, int dx, int dy, unsigned int mode, bool hierarchical, bool resolve_holes, bool min_coherence)

Description: Sizing operation on shapes from layouts

layout:	The layout from which to take the shapes
cell:	The cell (in 'layout') from which to take the shapes
layer:	The cell (in 'layout') from which to take the shapes
out:	The shapes container where to put the shapes into (is cleared before)
dx:	The sizing value in x-direction (see EdgeProcessor)
dy:	The sizing value in y-direction (see EdgeProcessor)
mode:	The sizing mode (see EdgeProcessor)
hierarchical:	Collect shapes from sub cells as well
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the sizing operation. This implementation takes shapes from a layout cell (optionally all in hierarchy) and produces new shapes in a shapes container.

(2) Signature: void **size** (const Layout layout, const Cell cell, unsigned int layer, Shapes out, int d, unsigned int mode, bool hierarchical, bool resolve_holes, bool min_coherence)

Description: Sizing operation on shapes from layouts

layout:	The layout from which to take the shapes
cell:	The cell (in 'layout') from which to take the shapes
layer:	The cell (in 'layout') from which to take the shapes
out:	The shapes container where to put the shapes into (is cleared before)
d:	The sizing value (see EdgeProcessor)
mode:	The sizing mode (see EdgeProcessor)
hierarchical:	Collect shapes from sub cells as well
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the sizing operation. This implementation takes shapes from a layout cell (optionally all in hierarchy) and produces new shapes in a shapes container. This is the isotropic version which does not allow specification of different sizing values in x and y-direction.

(3) Signature: Edge[] **size** (Shape[] in, CplxTrans[] trans, int d, unsigned int mode)

Description: Size the given shapes

in:	The set of shapes to size
trans:	A corresponding set of transformations to apply on the shapes
d:	The sizing value
mode:	The sizing mode (see EdgeProcessor)

See the EdgeProcessor for a description of the sizing method. This implementation takes shapes rather than polygons for input and produces an edge set. This is isotropic version that does not allow to specify different values in x and y direction.

(4) Signature: Edge[] **size** (Shape[] in, CplxTrans[] trans, int dx, int dy, unsigned int mode)

Description: Size the given shapes

in:	The set of shapes to size
trans:	A corresponding set of transformations to apply on the shapes
dx:	The sizing value in x-direction
dy:	The sizing value in y-direction
mode:	The sizing mode (see EdgeProcessor)

See the EdgeProcessor for a description of the sizing method. This implementation takes shapes rather than polygons for input and produces an edge set.

(5) Signature: Edge[] **size** (Shape[] in, int d, unsigned int mode)

Description: Size the given shapes

in:	The set of shapes to size
d:	The sizing value
mode:	The sizing mode (see EdgeProcessor)

See the EdgeProcessor for a description of the sizing method. This implementation takes shapes rather than polygons for input and produces an edge set. This is isotropic version that does not allow to specify different values in x and y direction. This version does not feature a transformation for each shape (unity is assumed).

(6) Signature: Edge[] **size** (Shape[] in, int dx, int dy, unsigned int mode)

Description: Size the given shapes

in:	The set of shapes to size
dx:	The sizing value in x-direction
dy:	The sizing value in y-direction
mode:	The sizing mode (see EdgeProcessor)

See the EdgeProcessor for a description of the sizing method. This implementation takes shapes rather than polygons for input and produces an edge set.

This version does not feature a transformation for each shape (unity is assumed).

size_to_polygon

(1) Signature: Polygon[] **size_to_polygon** (Shape[] in, CplxTrans[] trans, int d, unsigned int mode, bool resolve_holes, bool min_coherence)

Description: Size the given shapes

in:	The set of shapes to size
trans:	A corresponding set of transformations to apply on the shapes
d:	The sizing value
mode:	The sizing mode (see EdgeProcessor)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the sizing method. This implementation takes shapes rather than polygons for input and produces a polygon set. This is isotropic version that does not allow to specify different values in x and y direction.

(2) Signature: Polygon[] **size_to_polygon** (Shape[] in, CplxTrans[] trans, int dx, int dy, unsigned int mode, bool resolve_holes, bool min_coherence)

Description: Size the given shapes

in:	The set of shapes to size
trans:	A corresponding set of transformations to apply on the shapes
dx:	The sizing value in x-direction
dy:	The sizing value in y-direction
mode:	The sizing mode (see EdgeProcessor)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the sizing method. This implementation takes shapes rather than polygons for input and produces a polygon set.

(3) Signature: Polygon[] **size_to_polygon** (Shape[] in, int d, unsigned int mode, bool resolve_holes, bool min_coherence)

Description: Size the given shapes

in:	The set of shapes to size

d:	The sizing value
mode:	The sizing mode (see EdgeProcessor)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the sizing method. This implementation takes shapes rather than polygons for input and produces a polygon set. This is isotropic version that does not allow to specify different values in x and y direction. This version does not feature a transformation for each shape (unity is assumed).

(4) Signature: Polygon[] **size_to_polygon** (Shape[] in, int dx, int dy, unsigned int mode, bool resolve_holes, bool min_coherence)

Description: Size the given shapes

in:	The set of shapes to size
dx:	The sizing value in x-direction
dy:	The sizing value in y-direction
mode:	The sizing mode (see EdgeProcessor)
resolve_holes:	true, if holes should be resolved into the hull
min_coherence:	true, if minimum polygons should be created for touching corners

See the EdgeProcessor for a description of the sizing method. This implementation takes shapes rather than polygons for input and produces a polygon set.

This version does not feature a transformation for each shape (unity is assumed).

4.96. API reference - Class Shapes

Notation used in Ruby API documentation

Module: db

Description: A collection of shapes

A shapes collection is a collection of geometrical objects, such as polygons, boxes, paths, edges, edge pairs or text objects.

Shapes objects are the basic containers for geometrical objects of a cell. Inside a cell, there is one Shapes object per layer.

Public constructors

new Shapes ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Shapes other)	Assigns another object to self
[const]	Cell ptr	**cell**		Gets the cell the shape container belongs to
	void	**clear**		Clears the shape container
	void	**clear**	(unsigned int flags)	Clears certain shape types from the shape container
[const]	new Shapes ptr	**dup**		Creates a copy of self
[const,iter]	Shape	**each**	(unsigned int flags)	Gets all shapes
[const,ite	Shape	**each**		Gets all shapes
[const,iter]	Shape	**each_overlapping**	(unsigned int flags, const Box region)	Gets all shapes that overlap the search box (region)
[const,ite	Shape	**each_overlapping**	(unsigned int flags, const DBox region)	Gets all shapes that overlap the search box (region) where the search box is given in micrometer units

[const,iter]	Shape	**each_overlapping**	(const Box region)	Gets all shapes that overlap the search box (region)
[const,ite	Shape	**each_overlapping**	(const DBox region)	Gets all shapes that overlap the search box (region) where the search box is given in micrometer units
[const,iter]	Shape	**each_touching**	(unsigned int flags, const Box region)	Gets all shapes that touch the search box (region)
[const,ite	Shape	**each_touching**	(unsigned int flags, const DBox region)	Gets all shapes that touch the search box (region) where the search box is given in micrometer units
[const,iter]	Shape	**each_touching**	(const Box region)	Gets all shapes that touch the search box (region)
[const,ite	Shape	**each_touching**	(const DBox region)	Gets all shapes that touch the search box (region) where the search box is given in micrometer units
	void	**erase**	(const Shape shape)	Erases the shape pointed to by the given Shape object
[const]	Shape	**find**	(const Shape shape)	Finds a shape inside this collected
	Shape	**insert**	(const Shape shape)	Inserts a shape from a shape reference into the shapes list
	Shape	**insert**	(const Shape shape, const Trans trans)	Inserts a shape from a shape reference into the shapes list with a transformation
	Shape	**insert**	(const Shape shape, const DTrans trans)	Inserts a shape from a shape reference into the shapes list with a transformation (given in micrometer units)
	Shape	**insert**	(const Shape shape, const ICplxTrans trans)	Inserts a shape from a shape reference into the shapes list with a complex integer transformation
	Shape	**insert**	(const Shape shape, const DCplxTrans trans)	Inserts a shape from a shape reference into the shapes list with a complex integer transformation (given in micrometer units)
	void	**insert**	(const RecursiveShapeIterator iter)	Inserts the shapes taken from a recursive shape iterator
	void	**insert**	(const RecursiveShapeIterator iter, const ICplxTrans trans)	Inserts the shapes taken from a recursive shape iterator with a transformation
	void	**insert**	(const Shapes shapes)	Inserts the shapes taken from another shape container
	void	**insert**	(const Shapes shapes, const ICplxTrans trans)	Inserts the shapes taken from another shape container with a transformation
	void	**insert**	(const Shapes shapes, unsigned int flags)	Inserts the shapes taken from another shape container

void	insert	(const Shapes shapes, unsigned int flags, const ICplxTrans trans)	Inserts the shapes taken from another shape container with a transformation
void	insert	(const Region region)	Inserts the polygons from the region into this shape container
void	insert	(const Region region, const ICplxTrans trans)	Inserts the polygons from the region into this shape container with a transformation
void	insert	(const Region region, const DCplxTrans trans)	Inserts the polygons from the region into this shape container with a transformation (given in micrometer units)
void	insert	(const Edges edges)	Inserts the edges from the edge collection into this shape container
void	insert	(const Edges edges, const ICplxTrans trans)	Inserts the edges from the edge collection into this shape container with a transformation
void	insert	(const Edges edges, const DCplxTrans trans)	Inserts the edges from the edge collection into this shape container with a transformation (given in micrometer units)
void	insert	(const EdgePairs edge_pairs)	Inserts the edges from the edge pair collection into this shape container
void	insert	(const EdgePairs edge_pairs, const ICplxTrans trans)	Inserts the edge pairs from the edge pair collection into this shape container with a transformation
void	insert	(const EdgePairs edge_pairs, const DCplxTrans trans)	Inserts the edge pairs from the edge pair collection into this shape container with a transformation (given in micrometer units)
void	insert	(const Texts texts)	Inserts the texts from the text collection into this shape container
void	insert	(const Texts texts, const ICplxTrans trans)	Inserts the texts from the text collection into this shape container with a transformation
void	insert	(const Texts texts, const DCplxTrans trans)	Inserts the texts from the text collection into this shape container with a transformation (given in micrometer units)
Shape	insert	(const Box box)	Inserts a box into the shapes list
Shape	insert	(const DBox box)	Inserts a micrometer-unit box into the shapes list
Shape	insert	(const Path path)	Inserts a path into the shapes list
Shape	insert	(const DPath path)	Inserts a micrometer-unit path into the shapes list
Shape	insert	(const Edge edge)	Inserts an edge into the shapes list
Shape	insert	(const DEdge edge)	Inserts a micrometer-unit edge into the shapes list

Shape	insert	(const EdgePair edge_pair)	Inserts an edge pair into the shapes list
Shape	insert	(const DEdgePair edge_pair)	Inserts a micrometer-unit edge pair into the shapes list
Shape	insert	(const Point point)	Inserts an point into the shapes list
Shape	insert	(const DPoint point)	Inserts a micrometer-unit point into the shapes list
Shape	insert	(const Text text)	Inserts a text into the shapes list
Shape	insert	(const DText text)	Inserts a micrometer-unit text into the shapes list
Shape	insert	(const SimplePolygon simple_polygon)	Inserts a simple polygon into the shapes list
Shape	insert	(const DSimplePolygon simple_polygon)	Inserts a micrometer-unit simple polygon into the shapes list
Shape	insert	(const Polygon polygon)	Inserts a polygon into the shapes list
Shape	insert	(const DPolygon polygon)	Inserts a micrometer-unit polygon into the shapes list
Shape	insert	(const Box box, unsigned long property_id)	Inserts a box with properties into the shapes list
Shape	insert	(const DBox box, unsigned long property_id)	Inserts a micrometer-unit box with properties into the shapes list
Shape	insert	(const Path path, unsigned long property_id)	Inserts a path with properties into the shapes list
Shape	insert	(const DPath path, unsigned long property_id)	Inserts a micrometer-unit path with properties into the shapes list
Shape	insert	(const Edge edge, unsigned long property_id)	Inserts an edge with properties into the shapes list
Shape	insert	(const DEdge edge, unsigned long property_id)	Inserts a micrometer-unit edge with properties into the shapes list
Shape	insert	(const EdgePair edge_pair, unsigned long property_id)	Inserts an edge pair with properties into the shapes list
Shape	insert	(const DEdgePair edge_pair, unsigned long property_id)	Inserts a micrometer-unit edge pair with properties into the shapes list
Shape	insert	(const Text text, unsigned long property_id)	Inserts a text with properties into the shapes list
Shape	insert	(const DText text, unsigned long property_id)	Inserts a micrometer-unit text with properties into the shapes list
Shape	insert	(const SimplePolygon simple_polygon, unsigned long property_id)	Inserts a simple polygon with properties into the shapes list

	Shape	**insert**	(const DSimplePolygon simple_polygon, unsigned long property_id)	Inserts a micrometer-unit simple polygon with properties into the shapes list
	Shape	**insert**	(const Polygon polygon, unsigned long property_id)	Inserts a polygon with properties into the shapes list
	Shape	**insert**	(const DPolygon polygon, unsigned long property_id)	Inserts a micrometer-unit polygon with properties into the shapes list
	void	**insert_as_edges**	(const EdgePairs edge_pairs)	Inserts the edge pairs from the edge pair collection as individual edges into this shape container
	void	**insert_as_edges**	(const EdgePairs edge_pairs, const ICplxTrans trans)	Inserts the edge pairs from the edge pair collection as individual into this shape container with a transformation
	void	**insert_as_edges**	(const EdgePairs edge_pairs, const DCplxTrans trans)	Inserts the edge pairs from the edge pair collection as individual into this shape container with a transformation (given in micrometer units)
	void	**insert_as_polygons**	(const EdgePairs edge_pairs, int e)	Inserts the edge pairs from the edge pair collection as polygons into this shape container
	void	**insert_as_polygor**	(const EdgePairs edge_pairs, double e)	Inserts the edge pairs from the edge pair collection as polygons into this shape container
	void	**insert_as_polygons**	(const EdgePairs edge_pairs, const ICplxTrans e, int trans)	Inserts the edge pairs from the edge pair collection as polygons into this shape container with a transformation
	void	**insert_as_polygor**	(const EdgePairs edge_pairs, const DCplxTrans e, double trans)	Inserts the edge pairs from the edge pair collection as polygons into this shape container with a transformation
[const]	bool	**is_empty?**		Returns a value indicating whether the shapes container is empty
[const]	bool	**is_valid?**	(const Shape shape)	Tests if the given Shape object is still pointing to a valid object
	Layout ptr	**layout**		Gets the layout object the shape container belongs to
	Shape	**replace**	(const Shape shape, const Box box)	Replaces the given shape with a box
	Shape	**replace**	(const Shape shape, const DBox box)	Replaces the given shape with a box given in micrometer units
	Shape	**replace**	(const Shape shape, const Path path)	Replaces the given shape with a path
	Shape	**replace**	(const Shape shape, const DPath path)	Replaces the given shape with a path given in micrometer units

	Shape	**replace**	(const Shape shape, const Edge edge)	Replaces the given shape with an edge object
	Shape	**replace**	(const Shape shape, const DEdge edge)	Replaces the given shape with an edge given in micrometer units
	Shape	**replace**	(const Shape shape, const EdgePair edge_pair)	Replaces the given shape with an edge pair object
	Shape	**replace**	(const Shape shape, const DEdgePair edge_pair)	Replaces the given shape with an edge pair given in micrometer units
	Shape	**replace**	(const Shape shape, const Point point)	Replaces the given shape with an point object
	Shape	**replace**	(const Shape shape, const DPoint point)	Replaces the given shape with an point given in micrometer units
	Shape	**replace**	(const Shape shape, const Text text)	Replaces the given shape with a text object
	Shape	**replace**	(const Shape shape, const DText text)	Replaces the given shape with a text given in micrometer units
	Shape	**replace**	(const Shape shape, const SimplePolygon simple_polygon)	Replaces the given shape with a simple polygon
	Shape	**replace**	(const Shape shape, const DSimplePolygon simple_polygon)	Replaces the given shape with a simple polygon given in micrometer units
	Shape	**replace**	(const Shape shape, const Polygon polygon)	Replaces the given shape with a polygon
	Shape	**replace**	(const Shape shape, const DPolygon polygon)	Replaces the given shape with a polygon given in micrometer units
	Shape	**replace_prop_id**	(const Shape shape, unsigned long property_id)	Replaces (or install) the properties of a shape
[const]	unsigned long	**size**		Gets the number of shapes in this container
	void	**transform**	(const Trans trans)	Transforms all shapes with the given transformation
	void	**transform**	(const DTrans trans)	Transforms all shapes with the given transformation (given in micrometer units)
	void	**transform**	(const ICplxTrans trans)	Transforms all shapes with the given complex integer transformation
	void	**transform**	(const DCplxTrans trans)	Transforms all shapes with the given transformation (given in micrometer units)
	Shape	**transform**	(const Shape shape, const Trans trans)	Transforms the shape given by the reference with the given transformation

Shape	**transform**	(const Shape shape, const DTrans trans)	Transforms the shape given by the reference with the given transformation, where the transformation is given in micrometer units
Shape	**transform**	(const Shape shape, const ICplxTrans trans)	Transforms the shape given by the reference with the given complex integer space transformation
Shape	**transform**	(const Shape shape, const DCplxTrans trans)	Transforms the shape given by the reference with the given complex transformation, where the transformation is given in micrometer units

Public static methods and constants

unsigned int	**SAll**	Indicates that all shapes shall be retrieved
unsigned int	**SAllWithProperties**	Indicates that all shapes with properties shall be retrieved
unsigned int	**SBoxes**	Indicates that boxes shall be retrieved
unsigned int	**SEdgePairs**	Indicates that edge pairs shall be retrieved
unsigned int	**SEdges**	Indicates that edges shall be retrieved
unsigned int	**SPaths**	Indicates that paths shall be retrieved
unsigned int	**SPoints**	Indicates that points shall be retrieved
unsigned int	**SPolygons**	Indicates that polygons shall be retrieved
unsigned int	**SProperties**	Indicates that only shapes with properties shall be retrieved
unsigned int	**SRegions**	Indicates that objects which can be polygonized shall be retrieved (paths, boxes, polygons etc.)
unsigned int	**STexts**	Indicates that texts be retrieved
unsigned int	**SUserObjects**	Indicates that user objects shall be retrieved

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
	Shape	**insert_box**	(const Box box)	Use of this method is deprecated. Use insert instead
	Shape	**insert_box_with_properti**	(const Box box,	Use of this method is deprecated. Use insert instead

			unsigned long property_id)	
	Shape	**insert_edge**	(const Edge edge)	Use of this method is deprecated. Use insert instead
	Shape	**insert_edge_with_prope**	(const Edge edge, unsigned long property_id)	Use of this method is deprecated. Use insert instead
	Shape	**insert_path**	(const Path path)	Use of this method is deprecated. Use insert instead
	Shape	**insert_path_with_proper**	(const Path path, unsigned long property_id)	Use of this method is deprecated. Use insert instead
	Shape	**insert_point**	(const Point point)	Use of this method is deprecated. Use insert instead
	Shape	**insert_polygon**	(const Polygon polygon)	Use of this method is deprecated. Use insert instead
	Shape	**insert_polygon_with_properties**	(const Polygon polygon, unsigned long property_id)	Use of this method is deprecated. Use insert instead
	Shape	**insert_simple_polygon**	(const SimplePolygon simple_polygon)	Use of this method is deprecated. Use insert instead
	Shape	**insert_simple_polygon_with_properties**	(const SimplePolygon simple_polygon, unsigned long property_id)	Use of this method is deprecated. Use insert instead
	Shape	**insert_text**	(const Text text)	Use of this method is deprecated. Use insert instead
	Shape	**insert_text_with_properties**	(const Text text, unsigned long property_id)	Use of this method is deprecated. Use insert instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[static]	unsigned int	**s_all**		Use of this method is deprecated. Use SAll instead
[static]	unsigned int	**s_all_with_properties**		Use of this method is deprecated. Use SAllWithProperties instead
[static]	unsigned int	**s_boxes**		Use of this method is deprecated. Use SBoxes instead

[static]	unsigned int	**s_edge_pairs**	Use of this method is deprecated. Use SEdgePairs instead
[static]	unsigned int	**s_edges**	Use of this method is deprecated. Use SEdges instead
[static]	unsigned int	**s_paths**	Use of this method is deprecated. Use SPaths instead
[static]	unsigned int	**s_points**	Use of this method is deprecated. Use SPoints instead
[static]	unsigned int	**s_polygons**	Use of this method is deprecated. Use SPolygons instead
[static]	unsigned int	**s_properties**	Use of this method is deprecated. Use SProperties instead
[static]	unsigned int	**s_regions**	Use of this method is deprecated. Use SRegions instead
[static]	unsigned int	**s_texts**	Use of this method is deprecated. Use STexts instead
[static]	unsigned int	**s_user_objects**	Use of this method is deprecated. Use SUserObjects instead

Detailed description

SAll

Signature: *[static]* unsigned int **SAll**

Description: Indicates that all shapes shall be retrieved

You can use this constant to construct 'except' classes - e.g. to specify 'all shape types except boxes' use

```
SAll - SBoxes
```

Python specific notes:
The object exposes a readable attribute 'SAll'. This is the getter.

SAllWithProperties

Signature: *[static]* unsigned int **SAllWithProperties**

Description: Indicates that all shapes with properties shall be retrieved

Using this selector means to retrieve only shapes with properties.You can use this constant to construct 'except' classes - e.g. to specify 'all shape types with properties except boxes' use

```
SAllWithProperties - SBoxes
```

Python specific notes:
The object exposes a readable attribute 'SAllWithProperties'. This is the getter.

SBoxes

Signature: *[static]* unsigned int **SBoxes**

Description: Indicates that boxes shall be retrieved

Python specific notes:

	The object exposes a readable attribute 'SBoxes'. This is the getter.
SEdgePairs	**Signature**: *[static]* unsigned int **SEdgePairs** **Description**: Indicates that edge pairs shall be retrieved **Python specific notes:** The object exposes a readable attribute 'SEdgePairs'. This is the getter.
SEdges	**Signature**: *[static]* unsigned int **SEdges** **Description**: Indicates that edges shall be retrieved **Python specific notes:** The object exposes a readable attribute 'SEdges'. This is the getter.
SPaths	**Signature**: *[static]* unsigned int **SPaths** **Description**: Indicates that paths shall be retrieved **Python specific notes:** The object exposes a readable attribute 'SPaths'. This is the getter.
SPoints	**Signature**: *[static]* unsigned int **SPoints** **Description**: Indicates that points shall be retrieved This constant has been added in version 0.28. **Python specific notes:** The object exposes a readable attribute 'SPoints'. This is the getter.
SPolygons	**Signature**: *[static]* unsigned int **SPolygons** **Description**: Indicates that polygons shall be retrieved **Python specific notes:** The object exposes a readable attribute 'SPolygons'. This is the getter.
SProperties	**Signature**: *[static]* unsigned int **SProperties** **Description**: Indicates that only shapes with properties shall be retrieved You can or-combine this flag with the plain shape types to select a certain shape type, but only those shapes with properties. For example to select boxes with properties, use 'SProperties | SBoxes'. **Python specific notes:** The object exposes a readable attribute 'SProperties'. This is the getter.
SRegions	**Signature**: *[static]* unsigned int **SRegions** **Description**: Indicates that objects which can be polygonized shall be retrieved (paths, boxes, polygons etc.) This constant has been added in version 0.27. **Python specific notes:** The object exposes a readable attribute 'SRegions'. This is the getter.
STexts	**Signature**: *[static]* unsigned int **STexts** **Description**: Indicates that texts be retrieved **Python specific notes:**

The object exposes a readable attribute 'STexts'. This is the getter.

SUserObjects

Signature: *[static]* unsigned int **SUserObjects**

Description: Indicates that user objects shall be retrieved

Python specific notes:
The object exposes a readable attribute 'SUserObjects'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Shapes other)

Description: Assigns another object to self

cell

Signature: *[const]* Cell ptr **cell**

Description: Gets the cell the shape container belongs to

This method returns nil if the shape container does not belong to a cell.

This method has been added in version 0.28.

clear

(1) Signature: void **clear**

Description: Clears the shape container

This method has been introduced in version 0.16.

(2) Signature: void **clear** (unsigned int flags)

Description: Clears certain shape types from the shape container

Only shapes matching the shape types from 'flags' are removed. 'flags' is a combination of the S... constants.

This method has been introduced in version 0.28.9.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Shapes ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each

(1) Signature: *[const,iter]* Shape **each** (unsigned int flags)

Description: Gets all shapes

flags: An "or"-ed combination of the S... constants

(2) Signature: *[const,iter]* Shape **each**

Description: Gets all shapes

This call is equivalent to each(SAll). This convenience method has been introduced in version 0.16

Python specific notes:
This method enables iteration of the object.

each_overlapping

(1) Signature: *[const,iter]* Shape **each_overlapping** (unsigned int flags, const Box region)

Description: Gets all shapes that overlap the search box (region)

flags: An "or"-ed combination of the S... constants

region: The rectangular search region

This method was introduced in version 0.16

(2) Signature: *[const,iter]* Shape **each_overlapping** (unsigned int flags, const DBox region)

Description: Gets all shapes that overlap the search box (region) where the search box is given in micrometer units

flags: An "or"-ed combination of the S... constants

region: The rectangular search region as a DBox object in micrometer units

This method was introduced in version 0.25

(3) Signature: *[const,iter]* Shape **each_overlapping** (const Box region)

Description: Gets all shapes that overlap the search box (region)

region: The rectangular search region

This call is equivalent to each_overlapping(SAll,region). This convenience method has been introduced in version 0.16

(4) Signature: *[const,iter]* Shape **each_overlapping** (const DBox region)

Description: Gets all shapes that overlap the search box (region) where the search box is given in micrometer units

region: The rectangular search region as a DBox object in micrometer units

This call is equivalent to each_touching(SAll,region).

This method was introduced in version 0.25

each_touching

(1) Signature: *[const,iter]* Shape **each_touching** (unsigned int flags, const Box region)

Description: Gets all shapes that touch the search box (region)

flags: An "or"-ed combination of the S... constants

region: The rectangular search region

This method was introduced in version 0.16

(2) Signature: *[const,iter]* Shape **each_touching** (unsigned int flags, const DBox region)

Description: Gets all shapes that touch the search box (region) where the search box is given in micrometer units

flags:	An "or"-ed combination of the S... constants
region:	The rectangular search region as a DBox object in micrometer units

This method was introduced in version 0.25

(3) Signature: *[const,iter]* Shape **each_touching** (const Box region)

Description: Gets all shapes that touch the search box (region)

region:	The rectangular search region

This call is equivalent to each_touching(SAll,region). This convenience method has been introduced in version 0.16

(4) Signature: *[const,iter]* Shape **each_touching** (const DBox region)

Description: Gets all shapes that touch the search box (region) where the search box is given in micrometer units

region:	The rectangular search region as a DBox object in micrometer units

This call is equivalent to each_touching(SAll,region).

This method was introduced in version 0.25

erase

Signature: void **erase** (const Shape shape)

Description: Erases the shape pointed to by the given Shape object

shape:	The shape which to destroy

This method has been introduced in version 0.16. It can only be used in editable mode. Erasing a shape will invalidate the shape reference. Access to this reference may then render invalid results.

find

Signature: *[const]* Shape **find** (const Shape shape)

Description: Finds a shape inside this collected

This method has been introduced in version 0.21. This method tries to find the given shape in this collection. The original shape may be located in another collection. If the shape is found, this method returns a reference to the shape in this collection, otherwise a null reference is returned.

insert

(1) Signature: Shape **insert** (const Shape shape)

Description: Inserts a shape from a shape reference into the shapes list

Returns:	A reference (a Shape object) to the newly created shape

This method has been introduced in version 0.16.

(2) Signature: Shape **insert** (const Shape shape, const Trans trans)

Description: Inserts a shape from a shape reference into the shapes list with a transformation

shape:	The shape to insert
trans:	The transformation to apply before the shape is inserted
Returns:	A reference (a Shape object) to the newly created shape

This method has been introduced in version 0.22.

(3) Signature: Shape **insert** (const Shape shape, const DTrans trans)

Description: Inserts a shape from a shape reference into the shapes list with a transformation (given in micrometer units)

shape:	The shape to insert
trans:	The transformation to apply before the shape is inserted (displacement in micrometers)
Returns:	A reference (a Shape object) to the newly created shape

This method has been introduced in version 0.25.

(4) Signature: Shape **insert** (const Shape shape, const ICplxTrans trans)

Description: Inserts a shape from a shape reference into the shapes list with a complex integer transformation

shape:	The shape to insert
trans:	The transformation to apply before the shape is inserted
Returns:	A reference (a Shape object) to the newly created shape

This method has been introduced in version 0.22.

(5) Signature: Shape **insert** (const Shape shape, const DCplxTrans trans)

Description: Inserts a shape from a shape reference into the shapes list with a complex integer transformation (given in micrometer units)

shape:	The shape to insert
trans:	The transformation to apply before the shape is inserted (displacement in micrometer units)
Returns:	A reference (a Shape object) to the newly created shape

This method has been introduced in version 0.25.

(6) Signature: void **insert** (const RecursiveShapeIterator iter)

Description: Inserts the shapes taken from a recursive shape iterator

iter:	The iterator from which to take the shapes from

This method iterates over all shapes from the iterator and inserts them into the container.

This method has been introduced in version 0.25.3.

(7) Signature: void **insert** (const RecursiveShapeIterator iter, const ICplxTrans trans)

Description: Inserts the shapes taken from a recursive shape iterator with a transformation

iter:	The iterator from which to take the shapes from
trans:	The transformation to apply

This method iterates over all shapes from the iterator and inserts them into the container. The given transformation is applied before the shapes are inserted.

This method has been introduced in version 0.25.3.

(8) Signature: void **insert** (const Shapes shapes)

Description: Inserts the shapes taken from another shape container

shapes:	The other container from which to take the shapes from

This method takes all shapes from the given container and inserts them into this one.

This method has been introduced in version 0.25.3.

(9) Signature: void **insert** (const Shapes shapes, const ICplxTrans trans)

Description: Inserts the shapes taken from another shape container with a transformation

shapes:	The other container from which to take the shapes from
trans:	The transformation to apply

This method takes all shapes from the given container and inserts them into this one after applying the given transformation.

This method has been introduced in version 0.25.3.

(10) Signature: void **insert** (const Shapes shapes, unsigned int flags)

Description: Inserts the shapes taken from another shape container

shapes:	The other container from which to take the shapes from
flags:	The filter flags for taking the shapes from the input container (see S... constants)

This method takes all selected shapes from the given container and inserts them into this one.

This method has been introduced in version 0.25.3.

(11) Signature: void **insert** (const Shapes shapes, unsigned int flags, const ICplxTrans trans)

Description: Inserts the shapes taken from another shape container with a transformation

shapes:	The other container from which to take the shapes from
flags:	The filter flags for taking the shapes from the input container (see S... constants)
trans:	The transformation to apply

This method takes all selected shapes from the given container and inserts them into this one after applying the given transformation.

This method has been introduced in version 0.25.3.

(12) Signature: void **insert** (const Region region)

Description: Inserts the polygons from the region into this shape container

region:	The region to insert

This method inserts all polygons from the region into this shape container.

This method has been introduced in version 0.23.

(13) Signature: void **insert** (const Region region, const ICplxTrans trans)

Description: Inserts the polygons from the region into this shape container with a transformation

region:	The region to insert
trans:	The transformation to apply

This method inserts all polygons from the region into this shape container. Before a polygon is inserted, the given transformation is applied.

This method has been introduced in version 0.23.

(14) Signature: void **insert** (const Region region, const DCplxTrans trans)

Description: Inserts the polygons from the region into this shape container with a transformation (given in micrometer units)

region: The region to insert

trans: The transformation to apply (displacement in micrometer units)

This method inserts all polygons from the region into this shape container. Before a polygon is inserted, the given transformation is applied.

This method has been introduced in version 0.25.

(15) Signature: void **insert** (const Edges edges)

Description: Inserts the edges from the edge collection into this shape container

edges: The edges to insert

This method inserts all edges from the edge collection into this shape container.

This method has been introduced in version 0.23.

(16) Signature: void **insert** (const Edges edges, const ICplxTrans trans)

Description: Inserts the edges from the edge collection into this shape container with a transformation

edges: The edges to insert

trans: The transformation to apply

This method inserts all edges from the edge collection into this shape container. Before an edge is inserted, the given transformation is applied.

This method has been introduced in version 0.23.

(17) Signature: void **insert** (const Edges edges, const DCplxTrans trans)

Description: Inserts the edges from the edge collection into this shape container with a transformation (given in micrometer units)

edges: The edges to insert

trans: The transformation to apply (displacement in micrometer units)

This method inserts all edges from the edge collection into this shape container. Before an edge is inserted, the given transformation is applied.

This method has been introduced in version 0.25.

(18) Signature: void **insert** (const EdgePairs edge_pairs)

Description: Inserts the edges from the edge pair collection into this shape container

edges: The edge pairs to insert

This method inserts all edge pairs from the edge pair collection into this shape container.

This method has been introduced in version 0.26.

(19) Signature: void **insert** (const EdgePairs edge_pairs, const ICplxTrans trans)

Description: Inserts the edge pairs from the edge pair collection into this shape container with a transformation

edges: The edge pairs to insert

trans: The transformation to apply

This method inserts all edge pairs from the edge pair collection into this shape container. Before an edge pair is inserted, the given transformation is applied.

This method has been introduced in version 0.26.

(20) Signature: void **insert** (const EdgePairs edge_pairs, const DCplxTrans trans)

Description: Inserts the edge pairs from the edge pair collection into this shape container with a transformation (given in micrometer units)

edges: The edge pairs to insert

trans: The transformation to apply (displacement in micrometer units)

This method inserts all edge pairs from the edge pair collection into this shape container. Before an edge pair is inserted, the given transformation is applied.

This method has been introduced in version 0.26.

(21) Signature: void **insert** (const Texts texts)

Description: Inserts the texts from the text collection into this shape container

texts: The texts to insert

This method inserts all texts from the text collection into this shape container.

This method has been introduced in version 0.27.

(22) Signature: void **insert** (const Texts texts, const ICplxTrans trans)

Description: Inserts the texts from the text collection into this shape container with a transformation

edges: The texts to insert

trans: The transformation to apply

This method inserts all texts from the text collection into this shape container. Before an text is inserted, the given transformation is applied.

This method has been introduced in version 0.27.

(23) Signature: void **insert** (const Texts texts, const DCplxTrans trans)

Description: Inserts the texts from the text collection into this shape container with a transformation (given in micrometer units)

edges: The text to insert

trans: The transformation to apply (displacement in micrometer units)

This method inserts all texts from the text collection into this shape container. Before an text is inserted, the given transformation is applied.

This method has been introduced in version 0.27.

(24) Signature: Shape **insert** (const Box box)

Description: Inserts a box into the shapes list

Returns: A reference to the new shape (a Shape object)

Starting with version 0.16, this method returns a reference to the newly created shape

(25) Signature: Shape **insert** (const DBox box)

Description: Inserts a micrometer-unit box into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a Box argument, except that it will internally translate the box from micrometer to database units.

This variant has been introduced in version 0.25.

(26) Signature: Shape **insert** (const Path path)

Description: Inserts a path into the shapes list

Returns: A reference to the new shape (a Shape object)

Starting with version 0.16, this method returns a reference to the newly created shape

(27) Signature: Shape **insert** (const DPath path)

Description: Inserts a micrometer-unit path into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a Path argument, except that it will internally translate the path from micrometer to database units.

This variant has been introduced in version 0.25.

(28) Signature: Shape **insert** (const Edge edge)

Description: Inserts an edge into the shapes list

Starting with version 0.16, this method returns a reference to the newly created shape

(29) Signature: Shape **insert** (const DEdge edge)

Description: Inserts a micrometer-unit edge into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a Edge argument, except that it will internally translate the edge from micrometer to database units.

This variant has been introduced in version 0.25.

(30) Signature: Shape **insert** (const EdgePair edge_pair)

Description: Inserts an edge pair into the shapes list

This method has been introduced in version 0.26.

(31) Signature: Shape **insert** (const DEdgePair edge_pair)

Description: Inserts a micrometer-unit edge pair into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a EdgePair argument, except that it will internally translate the edge pair from micrometer to database units.

This variant has been introduced in version 0.26.

(32) Signature: Shape **insert** (const Point point)

Description: Inserts an point into the shapes list

This variant has been introduced in version 0.28.

(33) Signature: Shape **insert** (const DPoint point)

Description: Inserts a micrometer-unit point into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a Point argument, except that it will internally translate the point from micrometer to database units.

This variant has been introduced in version 0.28.

(34) Signature: Shape **insert** (const Text text)

Description: Inserts a text into the shapes list

Returns: A reference to the new shape (a Shape object)

Starting with version 0.16, this method returns a reference to the newly created shape

(35) Signature: Shape **insert** (const DText text)

Description: Inserts a micrometer-unit text into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a Text argument, except that it will internally translate the text from micrometer to database units.

This variant has been introduced in version 0.25.

(36) Signature: Shape **insert** (const SimplePolygon simple_polygon)

Description: Inserts a simple polygon into the shapes list

Returns: A reference to the new shape (a Shape object)

Starting with version 0.16, this method returns a reference to the newly created shape

(37) Signature: Shape **insert** (const DSimplePolygon simple_polygon)

Description: Inserts a micrometer-unit simple polygon into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a SimplePolygon argument, except that it will internally translate the polygon from micrometer to database units.

This variant has been introduced in version 0.25.

(38) Signature: Shape **insert** (const Polygon polygon)

Description: Inserts a polygon into the shapes list

Returns: A reference to the new shape (a Shape object)

Starting with version 0.16, this method returns a reference to the newly created shape

(39) Signature: Shape **insert** (const DPolygon polygon)

Description: Inserts a micrometer-unit polygon into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a Polygon argument, except that it will internally translate the polygon from micrometer to database units.

This variant has been introduced in version 0.25.

(40) Signature: Shape **insert** (const Box box, unsigned long property_id)

Description: Inserts a box with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape

(41) Signature: Shape **insert** (const DBox box, unsigned long property_id)

Description: Inserts a micrometer-unit box with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a Box argument and a property ID, except that it will internally translate the box from micrometer to database units.

This variant has been introduced in version 0.25.

(42) Signature: Shape **insert** (const Path path, unsigned long property_id)

Description: Inserts a path with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape

(43) Signature: Shape **insert** (const DPath path, unsigned long property_id)

Description: Inserts a micrometer-unit path with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a Path argument and a property ID, except that it will internally translate the path from micrometer to database units.

This variant has been introduced in version 0.25.

(44) Signature: Shape **insert** (const Edge edge, unsigned long property_id)

Description: Inserts an edge with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape.

(45) Signature: Shape **insert** (const DEdge edge, unsigned long property_id)

Description: Inserts a micrometer-unit edge with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a Edge argument and a property ID, except that it will internally translate the edge from micrometer to database units.

This variant has been introduced in version 0.25.

(46) Signature: Shape **insert** (const EdgePair edge_pair, unsigned long property_id)

Description: Inserts an edge pair with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. This method has been introduced in version 0.26.

(47) Signature: Shape **insert** (const DEdgePair edge_pair, unsigned long property_id)

Description: Inserts a micrometer-unit edge pair with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a EdgePair argument and a property ID, except that it will internally translate the edge pair from micrometer to database units.

This variant has been introduced in version 0.26.

(48) Signature: Shape **insert** (const Text text, unsigned long property_id)

Description: Inserts a text with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape

(49) Signature: Shape **insert** (const DText text, unsigned long property_id)

Description: Inserts a micrometer-unit text with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a Text argument and a property ID, except that it will internally translate the text from micrometer to database units.

This variant has been introduced in version 0.25.

(50) Signature: Shape **insert** (const SimplePolygon simple_polygon, unsigned long property_id)

Description: Inserts a simple polygon with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape

(51) Signature: Shape **insert** (const DSimplePolygon simple_polygon, unsigned long property_id)

Description: Inserts a micrometer-unit simple polygon with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a SimplePolygon argument and a property ID, except that it will internally translate the simple polygon from micrometer to database units.

This variant has been introduced in version 0.25.

(52) Signature: Shape **insert** (const Polygon polygon, unsigned long property_id)

Description: Inserts a polygon with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape

(53) Signature: Shape **insert** (const DPolygon polygon, unsigned long property_id)

Description: Inserts a micrometer-unit polygon with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

This method behaves like the insert version with a Polygon argument and a property ID, except that it will internally translate the polygon from micrometer to database units.

This variant has been introduced in version 0.25.

insert_as_edges

(1) Signature: void **insert_as_edges** (const EdgePairs edge_pairs)

Description: Inserts the edge pairs from the edge pair collection as individual edges into this shape container

edge_pairs: The edge pairs to insert

This method inserts all edge pairs from the edge pair collection into this shape container. Each edge from the edge pair is inserted individually into the shape container.

This method has been introduced in version 0.23.

(2) Signature: void **insert_as_edges** (const EdgePairs edge_pairs, const ICplxTrans trans)

Description: Inserts the edge pairs from the edge pair collection as individual into this shape container with a transformation

edges: The edge pairs to insert

trans: The transformation to apply

This method inserts all edge pairs from the edge pair collection into this shape container. Each edge from the edge pair is inserted individually into the shape container. Before each edge is inserted into the shape collection, the given transformation is applied.

This method has been introduced in version 0.23.

(3) Signature: void **insert_as_edges** (const EdgePairs edge_pairs, const DCplxTrans trans)

Description: Inserts the edge pairs from the edge pair collection as individual into this shape container with a transformation (given in micrometer units)

edges: The edge pairs to insert

trans: The transformation to apply (displacement in micrometer units)

This method inserts all edge pairs from the edge pair collection into this shape container. Each edge from the edge pair is inserted individually into the shape container. Before each edge is inserted into the shape collection, the given transformation is applied.

This method has been introduced in version 0.25.

insert_as_polygons

(1) Signature: void **insert_as_polygons** (const EdgePairs edge_pairs, int e)

Description: Inserts the edge pairs from the edge pair collection as polygons into this shape container

edge_pairs: The edge pairs to insert

e: The extension to apply when converting the edges to polygons (in database units)

This method inserts all edge pairs from the edge pair collection into this shape container. The edge pairs are converted to polygons covering the area between the edges. The extension parameter specifies a sizing which is applied when converting the edge pairs to polygons. This way, degenerated edge pairs (i.e. two point-like edges) do not vanish.

This method has been introduced in version 0.23.

(2) Signature: void **insert_as_polygons** (const EdgePairs edge_pairs, double e)

Description: Inserts the edge pairs from the edge pair collection as polygons into this shape container

edge_pairs: The edge pairs to insert

e: The extension to apply when converting the edges to polygons (in micrometer units)

This method is identical to the version with a integer-type e parameter, but for this version the e parameter is given in micrometer units.

This method has been introduced in version 0.25.

(3) Signature: void **insert_as_polygons** (const EdgePairs edge_pairs, const ICplxTrans e, int trans)

Description: Inserts the edge pairs from the edge pair collection as polygons into this shape container with a transformation

edges: The edge pairs to insert

e: The extension to apply when converting the edges to polygons (in database units)

trans: The transformation to apply

This method inserts all edge pairs from the edge pair collection into this shape container. The edge pairs are converted to polygons covering the area between the edges. The extension parameter specifies a sizing which is applied when converting the edge pairs to polygons. This way, degenerated edge pairs (i.e. two point-like edges) do not vanish. Before a polygon is inserted into the shape collection, the given transformation is applied.

This method has been introduced in version 0.23.

(4) Signature: void **insert_as_polygons** (const EdgePairs edge_pairs, const DCplxTrans e, double trans)

Description: Inserts the edge pairs from the edge pair collection as polygons into this shape container with a transformation

edges: The edge pairs to insert

e: The extension to apply when converting the edges to polygons (in micrometer units)

trans: The transformation to apply (displacement in micrometer units)

This method is identical to the version with a integer-type e and trans parameter, but for this version the e parameter is given in micrometer units and the trans parameter is a micrometer-unit transformation.

This method has been introduced in version 0.25.

insert_box

Signature: Shape **insert_box** (const Box box)

Description: Inserts a box into the shapes list

Returns: A reference to the new shape (a Shape object)

Use of this method is deprecated. Use insert instead

Starting with version 0.16, this method returns a reference to the newly created shape

insert_box_with_properties

Signature: Shape **insert_box_with_properties** (const Box box, unsigned long property_id)

Description: Inserts a box with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

Use of this method is deprecated. Use insert instead

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape

insert_edge

Signature: Shape **insert_edge** (const Edge edge)

Description: Inserts an edge into the shapes list

Use of this method is deprecated. Use insert instead

Starting with version 0.16, this method returns a reference to the newly created shape

insert_edge_with_properties

Signature: Shape **insert_edge_with_properties** (const Edge edge, unsigned long property_id)

Description: Inserts an edge with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

Use of this method is deprecated. Use insert instead

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape.

insert_path

Signature: Shape **insert_path** (const Path path)

Description: Inserts a path into the shapes list

Returns: A reference to the new shape (a Shape object)

Use of this method is deprecated. Use insert instead

Starting with version 0.16, this method returns a reference to the newly created shape

insert_path_with_properties

Signature: Shape **insert_path_with_properties** (const Path path, unsigned long property_id)

Description: Inserts a path with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

Use of this method is deprecated. Use insert instead

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape

insert_point

Signature: Shape **insert_point** (const Point point)

Description: Inserts an point into the shapes list

Use of this method is deprecated. Use insert instead

This variant has been introduced in version 0.28.

insert_polygon

Signature: Shape **insert_polygon** (const Polygon polygon)

Description: Inserts a polygon into the shapes list

Returns: A reference to the new shape (a Shape object)

Use of this method is deprecated. Use insert instead

Starting with version 0.16, this method returns a reference to the newly created shape

insert_polygon_with_properties

Signature: Shape **insert_polygon_with_properties** (const Polygon polygon, unsigned long property_id)

Description: Inserts a polygon with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

Use of this method is deprecated. Use insert instead

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape

insert_simple_polygon

Signature: Shape **insert_simple_polygon** (const SimplePolygon simple_polygon)

Description: Inserts a simple polygon into the shapes list

Returns: A reference to the new shape (a Shape object)

Use of this method is deprecated. Use insert instead

Starting with version 0.16, this method returns a reference to the newly created shape

insert_simple_polygon_with_properties

Signature: Shape **insert_simple_polygon_with_properties** (const SimplePolygon simple_polygon, unsigned long property_id)

Description: Inserts a simple polygon with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

Use of this method is deprecated. Use insert instead

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape

insert_text

Signature: Shape **insert_text** (const Text text)

Description: Inserts a text into the shapes list

Returns: A reference to the new shape (a Shape object)

Use of this method is deprecated. Use insert instead

Starting with version 0.16, this method returns a reference to the newly created shape

insert_text_with_properties

Signature: Shape **insert_text_with_properties** (const Text text, unsigned long property_id)

Description: Inserts a text with properties into the shapes list

Returns: A reference to the new shape (a Shape object)

Use of this method is deprecated. Use insert instead

The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. Starting with version 0.16, this method returns a reference to the newly created shape

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_empty?

Signature: *[const]* bool **is_empty?**

Description: Returns a value indicating whether the shapes container is empty

This method has been introduced in version 0.20.

is_valid?

Signature: *[const]* bool **is_valid?** (const Shape shape)

Description: Tests if the given Shape object is still pointing to a valid object

This method has been introduced in version 0.16. If the shape represented by the given reference has been deleted, this method returns false. If however, another shape has been inserted already that occupies the original shape's position, this method will return true again.

layout

Signature: Layout ptr **layout**

Description: Gets the layout object the shape container belongs to

This method returns nil if the shape container does not belong to a layout.

This method has been added in version 0.28.

new

Signature: *[static]* new Shapes ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

replace

(1) Signature: Shape **replace** (const Shape shape, const Box box)

Description: Replaces the given shape with a box

Returns: A reference to the new shape (a Shape object)

This method has been introduced with version 0.16. It replaces the given shape with the object specified. It does not change the property Id. To change the property Id, use the replace_prop_id method. To replace a shape and discard the property Id, erase the shape and insert a new shape. This method is permitted in editable mode only.

(2) Signature: Shape **replace** (const Shape shape, const DBox box)

Description: Replaces the given shape with a box given in micrometer units

Returns: A reference to the new shape (a Shape object)

This method behaves like the replace version with a Box argument, except that it will internally translate the box from micrometer to database units.

This variant has been introduced in version 0.25.

(3) Signature: Shape **replace** (const Shape shape, const Path path)

Description: Replaces the given shape with a path

Returns: A reference to the new shape (a Shape object)

This method has been introduced with version 0.16. It replaces the given shape with the object specified. It does not change the property Id. To change the property Id, use the replace_prop_id method. To replace a shape and discard the property Id, erase the shape and insert a new shape. This method is permitted in editable mode only.

(4) Signature: Shape **replace** (const Shape shape, const DPath path)

Description: Replaces the given shape with a path given in micrometer units

Returns: A reference to the new shape (a Shape object)

This method behaves like the replace version with a Path argument, except that it will internally translate the path from micrometer to database units.

This variant has been introduced in version 0.25.

(5) Signature: Shape **replace** (const Shape shape, const Edge edge)

Description: Replaces the given shape with an edge object

This method has been introduced with version 0.16. It replaces the given shape with the object specified. It does not change the property Id. To change the property Id, use the replace_prop_id method. To replace a shape and discard the property Id, erase the shape and insert a new shape. This method is permitted in editable mode only.

(6) Signature: Shape **replace** (const Shape shape, const DEdge edge)

Description: Replaces the given shape with an edge given in micrometer units

Returns: A reference to the new shape (a Shape object)

This method behaves like the replace version with an Edge argument, except that it will internally translate the edge from micrometer to database units.

This variant has been introduced in version 0.25.

(7) Signature: Shape **replace** (const Shape shape, const EdgePair edge_pair)

Description: Replaces the given shape with an edge pair object

It replaces the given shape with the object specified. It does not change the property Id. To change the property Id, use the replace_prop_id method. To replace a shape and discard the property Id, erase the shape and insert a new shape. This method is permitted in editable mode only.

This method has been introduced in version 0.26.

(8) Signature: Shape **replace** (const Shape shape, const DEdgePair edge_pair)

Description: Replaces the given shape with an edge pair given in micrometer units

Returns: A reference to the new shape (a Shape object)

This method behaves like the replace version with an EdgePair argument, except that it will internally translate the edge pair from micrometer to database units.

This variant has been introduced in version 0.26.

(9) Signature: Shape **replace** (const Shape shape, const Point point)

Description: Replaces the given shape with an point object

This method replaces the given shape with the object specified. It does not change the property Id. To change the property Id, use the replace_prop_id method. To replace a shape and discard the

property Id, erase the shape and insert a new shape. This variant has been introduced in version 0.28.

(10) Signature: Shape **replace** (const Shape shape, const DPoint point)

Description: Replaces the given shape with an point given in micrometer units

Returns: A reference to the new shape (a Shape object)

This method behaves like the replace version with an Point argument, except that it will internally translate the point from micrometer to database units.

This variant has been introduced in version 0.28.

(11) Signature: Shape **replace** (const Shape shape, const Text text)

Description: Replaces the given shape with a text object

Returns: A reference to the new shape (a Shape object)

This method has been introduced with version 0.16. It replaces the given shape with the object specified. It does not change the property Id. To change the property Id, use the replace_prop_id method. To replace a shape and discard the property Id, erase the shape and insert a new shape. This method is permitted in editable mode only.

(12) Signature: Shape **replace** (const Shape shape, const DText text)

Description: Replaces the given shape with a text given in micrometer units

Returns: A reference to the new shape (a Shape object)

This method behaves like the replace version with a Text argument, except that it will internally translate the text from micrometer to database units.

This variant has been introduced in version 0.25.

(13) Signature: Shape **replace** (const Shape shape, const SimplePolygon simple_polygon)

Description: Replaces the given shape with a simple polygon

Returns: A reference to the new shape (a Shape object)

This method has been introduced with version 0.16. It replaces the given shape with the object specified. It does not change the property Id. To change the property Id, use the replace_prop_id method. To replace a shape and discard the property Id, erase the shape and insert a new shape. This method is permitted in editable mode only.

(14) Signature: Shape **replace** (const Shape shape, const DSimplePolygon simple_polygon)

Description: Replaces the given shape with a simple polygon given in micrometer units

Returns: A reference to the new shape (a Shape object)

This method behaves like the replace version with a SimplePolygon argument, except that it will internally translate the simple polygon from micrometer to database units.

This variant has been introduced in version 0.25.

(15) Signature: Shape **replace** (const Shape shape, const Polygon polygon)

Description: Replaces the given shape with a polygon

Returns: A reference to the new shape (a Shape object)

This method has been introduced with version 0.16. It replaces the given shape with the object specified. It does not change the property Id. To change the property Id, use the replace_prop_id

method. To replace a shape and discard the property Id, erase the shape and insert a new shape. This method is permitted in editable mode only.

(16) Signature: Shape **replace** (const Shape shape, const DPolygon polygon)

Description: Replaces the given shape with a polygon given in micrometer units

Returns: A reference to the new shape (a Shape object)

This method behaves like the replace version with a Polygon argument, except that it will internally translate the polygon from micrometer to database units.

This variant has been introduced in version 0.25.

replace_prop_id

Signature: Shape **replace_prop_id** (const Shape shape, unsigned long property_id)

Description: Replaces (or install) the properties of a shape

Returns: A Shape object representing the new shape

This method has been introduced in version 0.16. It can only be used in editable mode. Changes the properties Id of the given shape or install a properties Id on that shape if it does not have one yet. The property Id must be obtained from the Layout object's property_id method which associates a property set with a property Id. This method will potentially invalidate the shape reference passed to it. Use the reference returned for future references.

s_all

Signature: *[static]* unsigned int **s_all**

Description: Indicates that all shapes shall be retrieved

Use of this method is deprecated. Use SAll instead

You can use this constant to construct 'except' classes - e.g. to specify 'all shape types except boxes' use

```
SAll - SBoxes
```

Python specific notes:
The object exposes a readable attribute 'SAll'. This is the getter.

s_all_with_properties

Signature: *[static]* unsigned int **s_all_with_properties**

Description: Indicates that all shapes with properties shall be retrieved

Use of this method is deprecated. Use SAllWithProperties instead

Using this selector means to retrieve only shapes with properties.You can use this constant to construct 'except' classes - e.g. to specify 'all shape types with properties except boxes' use

```
SAllWithProperties - SBoxes
```

Python specific notes:
The object exposes a readable attribute 'SAllWithProperties'. This is the getter.

s_boxes

Signature: *[static]* unsigned int **s_boxes**

Description: Indicates that boxes shall be retrieved

Use of this method is deprecated. Use SBoxes instead

Python specific notes:
The object exposes a readable attribute 'SBoxes'. This is the getter.

s_edge_pairs

Signature: *[static]* unsigned int **s_edge_pairs**

Description: Indicates that edge pairs shall be retrieved

Use of this method is deprecated. Use SEdgePairs instead

Python specific notes:
The object exposes a readable attribute 'SEdgePairs'. This is the getter.

s_edges

Signature: *[static]* unsigned int **s_edges**

Description: Indicates that edges shall be retrieved

Use of this method is deprecated. Use SEdges instead

Python specific notes:
The object exposes a readable attribute 'SEdges'. This is the getter.

s_paths

Signature: *[static]* unsigned int **s_paths**

Description: Indicates that paths shall be retrieved

Use of this method is deprecated. Use SPaths instead

Python specific notes:
The object exposes a readable attribute 'SPaths'. This is the getter.

s_points

Signature: *[static]* unsigned int **s_points**

Description: Indicates that points shall be retrieved

Use of this method is deprecated. Use SPoints instead

This constant has been added in version 0.28.

Python specific notes:
The object exposes a readable attribute 'SPoints'. This is the getter.

s_polygons

Signature: *[static]* unsigned int **s_polygons**

Description: Indicates that polygons shall be retrieved

Use of this method is deprecated. Use SPolygons instead

Python specific notes:
The object exposes a readable attribute 'SPolygons'. This is the getter.

s_properties

Signature: *[static]* unsigned int **s_properties**

Description: Indicates that only shapes with properties shall be retrieved

Use of this method is deprecated. Use SProperties instead

You can or-combine this flag with the plain shape types to select a certain shape type, but only those shapes with properties. For example to select boxes with properties, use 'SProperties | SBoxes'.

Python specific notes:
The object exposes a readable attribute 'SProperties'. This is the getter.

s_regions

Signature: *[static]* unsigned int **s_regions**

Description: Indicates that objects which can be polygonized shall be retrieved (paths, boxes, polygons etc.)

Use of this method is deprecated. Use SRegions instead

This constant has been added in version 0.27.

Python specific notes:

The object exposes a readable attribute 'SRegions'. This is the getter.

s_texts

Signature: *[static]* unsigned int **s_texts**

Description: Indicates that texts be retrieved

Use of this method is deprecated. Use STexts instead

Python specific notes:
The object exposes a readable attribute 'STexts'. This is the getter.

s_user_objects

Signature: *[static]* unsigned int **s_user_objects**

Description: Indicates that user objects shall be retrieved

Use of this method is deprecated. Use SUserObjects instead

Python specific notes:
The object exposes a readable attribute 'SUserObjects'. This is the getter.

size

Signature: *[const]* unsigned long **size**

Description: Gets the number of shapes in this container

Returns: The number of shapes in this container

This method was introduced in version 0.16

Python specific notes:
This method is also available as 'len(object)'.

transform

(1) Signature: void **transform** (const Trans trans)

Description: Transforms all shapes with the given transformation

This method will invalidate all references to shapes inside this collection.

It has been introduced in version 0.23.

(2) Signature: void **transform** (const DTrans trans)

Description: Transforms all shapes with the given transformation (given in micrometer units)

This method will invalidate all references to shapes inside this collection. The displacement of the transformation is given in micrometer units.

It has been introduced in version 0.25.

(3) Signature: void **transform** (const ICplxTrans trans)

Description: Transforms all shapes with the given complex integer transformation

This method will invalidate all references to shapes inside this collection.

It has been introduced in version 0.23.

(4) Signature: void **transform** (const DCplxTrans trans)

Description: Transforms all shapes with the given transformation (given in micrometer units)

This method will invalidate all references to shapes inside this collection. The displacement of the transformation is given in micrometer units.

It has been introduced in version 0.25.

(5) Signature: Shape **transform** (const Shape shape, const Trans trans)

Description: Transforms the shape given by the reference with the given transformation

Returns: A reference (a Shape object) to the new shape

The original shape may be deleted and re-inserted by this method. Therefore, a new reference is returned. It is permitted in editable mode only.

This method has been introduced in version 0.16.

(6) Signature: Shape **transform** (const Shape shape, const DTrans trans)

Description: Transforms the shape given by the reference with the given transformation, where the transformation is given in micrometer units

trans: The transformation to apply (displacement in micrometer units)

Returns: A reference (a Shape object) to the new shape

The original shape may be deleted and re-inserted by this method. Therefore, a new reference is returned. It is permitted in editable mode only. This method has been introduced in version 0.25.

(7) Signature: Shape **transform** (const Shape shape, const ICplxTrans trans)

Description: Transforms the shape given by the reference with the given complex integer space transformation

Returns: A reference (a Shape object) to the new shape

This method has been introduced in version 0.22. The original shape may be deleted and re-inserted by this method. Therefore, a new reference is returned. It is permitted in editable mode only.

(8) Signature: Shape **transform** (const Shape shape, const DCplxTrans trans)

Description: Transforms the shape given by the reference with the given complex transformation, where the transformation is given in micrometer units

trans: The transformation to apply (displacement in micrometer units)

Returns: A reference (a Shape object) to the new shape

The original shape may be deleted and re-inserted by this method. Therefore, a new reference is returned. It is permitted in editable mode only. This method has been introduced in version 0.25.

4.97. API reference - Class TechnologyComponent

Notation used in Ruby API documentation

Module: db

Description: A part of a technology definition

Technology components extend technology definitions (class Technology) by specialized subfeature definitions. For example, the net tracer supplies its technology-dependent specification through a technology component called NetTracerTechnology.

Components are managed within technologies and can be accessed from a technology using Technology#component.

This class has been introduced in version 0.25.

Public constructors

new TechnologyComponent ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.
[const]	string	**description**	Gets the human-readable description string of the technology component
[const]	string	**name**	Gets the formal name of the technology component

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

description

Signature: *[const]* string **description**

Description: Gets the human-readable description string of the technology component

destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
name	**Signature**: *[const]* string **name** **Description**: Gets the formal name of the technology component This is the name by which the component can be obtained from a technology using Technology#component.
new	**Signature**: *[static]* new TechnologyComponent ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.

4.98. API reference - Class Technology

Notation used in Ruby API documentation

Module: db

Description: Represents a technology

This class represents one technology from a set of technologies. The set of technologies available in the system can be obtained with technology_names. Individual technology definitions are returned with technology_by_name. Use create_technology to register new technologies and remove_technology to delete technologies.

Note that a Technology object needs to be registered in the system, before its name can be used to specify a technology, for example in Layout#technology_name. Technology objects created by create_technology are automatically registered. If you create a Technology object directly, you need to register it explicitly:

```
tech = RBA::Technology::new
tech.load("mytech.lyt")
RBA::Technology::register_technology(tech)
```

Note that in the latter example, an exception will be thrown if a technology with the same name already exists. Also note, that Technology#register will register a copy of the object, so modifying it after registration will not have any effect.

The Technology class has been introduced in version 0.25.

Public constructors

new Technology ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add_other_layers=**	(bool add)	Sets the flag indicating whether to add other layers to the layer properties
[const]	bool	**add_other_layers?**		Gets the flag indicating whether to add other layers to the layer properties
	void	**assign**	(const Technology other)	Assigns another object to self
[const]	string	**base_path**		Gets the base path of the technology

	TechnologyComponent ptr	**component**	(string name)	Gets the technology component with the given name
[const]	string[]	**component_names**		Gets the names of all components available for component
[const]	string	**correct_path**	(string path)	Makes a file path relative to the base path if one is specified
[const]	double	**dbu**		Gets the default database unit
	void	**dbu=**	(double dbu)	Sets the default database unit
[const]	string	**default_base_path**		Gets the default base path
[const]	double	**default_grid**		Gets the default grid
[const]	double[]	**default_grids**		Gets the default grids
	void	**default_grids=**	(double[] grids)	Sets the default grids
[const]	string	**description**		Gets the description
	void	**description=**	(string description)	Sets the description
[const]	new Technology ptr	**dup**		Creates a copy of self
[const]	string	**eff_layer_properties_file**		Gets the effective path of the layer properties file
[const]	string	**eff_path**	(string path)	Makes a file path relative to the base path if one is specified
[const]	string	**explicit_base_path**		Gets the explicit base path
	void	**explicit_base_path=**	(string path)	Sets the explicit base path
[const]	string	**group**		Gets the technology group
	void	**group=**	(string group)	Sets the technology group
[const]	string	**layer_properties_file**		Gets the path of the layer properties file
	void	**layer_properties_file=**	(string file)	Sets the path of the layer properties file
	void	**load**	(string file)	Loads the technology definition from a file
[const]	LoadLayoutOptions	**load_layout_options**		Gets the layout reader options
	void	**load_layout_options=**	(const LoadLayoutOptio options)	Sets the layout reader options
[const]	string	**name**		Gets the name of the technology
	void	**name=**	(string name)	Sets the name of the technology

[const]	void	**save**	(string file)	Saves the technology definition to a file
[const]	SaveLayoutOptions	**save_layout_options**		Gets the layout writer options
	void	**save_layout_options=**	(const SaveLayoutOptions options)	Sets the layout writer options
	void	**set_default_grids**	(double[] grids, double default_grid = 0)	Sets the default grids and the strong default one
[const]	string	**to_xml**		Returns a XML representation of this technolog

Public static methods and constants

void	**clear_technologies**		Clears all technologies
Technology ptr	**create_technology**	(string name)	Creates a new (empty) technology with the given name
bool	**has_technology?**	(string name)	Returns a value indicating whether there is a technology with this name
Technology ptr	**register_technology**	(const Technology tech)	Registers a technology in the system
void	**remove_technology**	(string name)	Removes the technology with the given name from the system
void	**technologies_from_xml**	(string xml)	Loads the technologies from a XML representation
string	**technologies_to_xml**		Returns a XML representation of all technologies registered in the system
Technology ptr	**technology_by_name**	(string name)	Gets the technology object for a given name
Technology	**technology_from_xml**	(string xml)	Loads the technology from a XML representation
string[]	**technology_names**		Gets a list of technology names defined in the system

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead

[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_other_layers=

Signature: void **add_other_layers=** (bool add)

Description: Sets the flag indicating whether to add other layers to the layer properties

Python specific notes:

	The object exposes a writable attribute 'add_other_layers'. This is the setter.
add_other_layers?	**Signature**: *[const]* bool **add_other_layers?** **Description**: Gets the flag indicating whether to add other layers to the layer properties **Python specific notes:** The object exposes a readable attribute 'add_other_layers'. This is the getter.
assign	**Signature**: void **assign** (const Technology other) **Description**: Assigns another object to self
base_path	**Signature**: *[const]* string **base_path** **Description**: Gets the base path of the technology The base path is the effective path where files are read from if their file path is a relative one. If the explicit path is set (see explicit_base_path=), it is used. If not, the default path is used. The default path is the one from which a technology file was imported. The explicit one is the one that is specified explicitly with explicit_base_path=.
clear_technologies	**Signature**: *[static]* void **clear_technologies** **Description**: Clears all technologies This method has been introduced in version 0.26.
component	**Signature**: TechnologyComponent ptr **component** (string name) **Description**: Gets the technology component with the given name The names are unique system identifiers. For all names, use component_names.
component_names	**Signature**: *[const]* string[] **component_names** **Description**: Gets the names of all components available for component
correct_path	**Signature**: *[const]* string **correct_path** (string path) **Description**: Makes a file path relative to the base path if one is specified This method turns an absolute path into one relative to the base path. Only files below the base path will be made relative. Files above or beside won't be made relative. See base_path for details about the default base path.
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
create_technology	**Signature**: *[static]* Technology ptr **create_technology** (string name) **Description**: Creates a new (empty) technology with the given name The new technology is already registered in the system.

This method returns a reference to the new technology.

dbu

Signature: *[const]* double **dbu**

Description: Gets the default database unit

The default database unit is the one used when creating a layout for example.

Python specific notes:
The object exposes a readable attribute 'dbu'. This is the getter.

dbu=

Signature: void **dbu=** (double dbu)

Description: Sets the default database unit

Python specific notes:
The object exposes a writable attribute 'dbu'. This is the setter.

default_base_path

Signature: *[const]* string **default_base_path**

Description: Gets the default base path

See base_path for details about the default base path.

Python specific notes:
The object exposes a readable attribute 'default_base_path'. This is the getter.

default_grid

Signature: *[const]* double **default_grid**

Description: Gets the default grid

The default grid is a specific one from the default grid list. It indicates the one that is taken if the current grid is not matching one of the default grids.

To set the default grid, use set_default_grids.

This property has been introduced in version 0.29.

default_grids

Signature: *[const]* double[] **default_grids**

Description: Gets the default grids

See default_grids for details.

This property has been introduced in version 0.28.17.

Python specific notes:
The object exposes a readable attribute 'default_grids'. This is the getter.

default_grids=

Signature: void **default_grids=** (double[] grids)

Description: Sets the default grids

If not empty, this list replaces the global grid list for this technology. Note that this method will reset the default grid (see default_grid). Use set_default_grids to set the default grids and the strong default one.

This property has been introduced in version 0.28.17.

Python specific notes:
The object exposes a writable attribute 'default_grids'. This is the setter.

description

Signature: *[const]* string **description**

Description: Gets the description

The technology description is shown to the user in technology selection dialogs and for display purposes.

Python specific notes:
The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string description)

Description: Sets the description

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Technology ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

eff_layer_properties_file

Signature: *[const]* string **eff_layer_properties_file**

Description: Gets the effective path of the layer properties file

eff_path

Signature: *[const]* string **eff_path** (string path)

Description: Makes a file path relative to the base path if one is specified

This method will return the actual path for a file from the file's path. If the input path is a relative one, it will be made absolute by using the base path.

See base_path for details about the default base path.

explicit_base_path

Signature: *[const]* string **explicit_base_path**

Description: Gets the explicit base path

See base_path for details about the explicit base path.

Python specific notes:
The object exposes a readable attribute 'explicit_base_path'. This is the getter.

explicit_base_path=

Signature: void **explicit_base_path=** (string path)

Description: Sets the explicit base path

See base_path for details about the explicit base path.

Python specific notes:
The object exposes a writable attribute 'explicit_base_path'. This is the setter.

group

Signature: *[const]* string **group**

Description: Gets the technology group

The technology group is used to group certain technologies together in the technology selection menu. Technologies with the same group are put under a submenu with that group title.

The 'group' attribute has been introduced in version 0.26.2.

Python specific notes:
The object exposes a readable attribute 'group'. This is the getter.

group=

Signature: void **group=** (string group)

Description: Sets the technology group

See group for details about this attribute.

The 'group' attribute has been introduced in version 0.26.2.

Python specific notes:
The object exposes a writable attribute 'group'. This is the setter.

has_technology?

Signature: *[static]* bool **has_technology?** (string name)

Description: Returns a value indicating whether there is a technology with this name

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

layer_properties_file

Signature: *[const]* string **layer_properties_file**

Description: Gets the path of the layer properties file

If empty, no layer properties file is associated with the technology. If non-empty, this path will be corrected by the base path (see correct_path) and this layer properties file will be loaded for layouts with this technology.

Python specific notes:
The object exposes a readable attribute 'layer_properties_file'. This is the getter.

layer_properties_file=

Signature: void **layer_properties_file=** (string file)

Description: Sets the path of the layer properties file

See layer_properties_file for details about this property.

Python specific notes:
The object exposes a writable attribute 'layer_properties_file'. This is the setter.

load

Signature: void **load** (string file)

Description: Loads the technology definition from a file

load_layout_options

Signature: *[const]* LoadLayoutOptions **load_layout_options**

Description: Gets the layout reader options

This method returns the layout reader options that are used when reading layouts with this technology.

Change the reader options by modifying the object and using the setter to change it:

```
opt = tech.load_layout_options
opt.dxf_dbu = 2.5
tech.load_layout_options = opt
```

Python specific notes:
The object exposes a readable attribute 'load_layout_options'. This is the getter.

load_layout_options=

Signature: void **load_layout_options=** (const LoadLayoutOptions options)

Description: Sets the layout reader options

See load_layout_options for a description of this property.

Python specific notes:
The object exposes a writable attribute 'load_layout_options'. This is the setter.

name

Signature: *[const]* string **name**

Description: Gets the name of the technology

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string name)

Description: Sets the name of the technology

Python specific notes:
The object exposes a writable attribute 'name'. This is the setter.

new

Signature: *[static]* new Technology ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

register_technology

Signature: *[static]* Technology ptr **register_technology** (const Technology tech)

Description: Registers a technology in the system

Only after a technology is registered, it can be used in the system, e.g. by specifying its name in Layout#technology_name. While create_technology already registers the technology, this method allows registering a Technology object that has created in other ways.

This method returns a reference to the new technology object, which is a copy of the argument. remove_technology can be used to remove a technology registered by this method.

This method has been introduced in version 0.28.14.

remove_technology

Signature: *[static]* void **remove_technology** (string name)

Description: Removes the technology with the given name from the system

save

Signature: *[const]* void **save** (string file)

Description: Saves the technology definition to a file

save_layout_options

Signature: *[const]* SaveLayoutOptions **save_layout_options**

Description: Gets the layout writer options

This method returns the layout writer options that are used when writing layouts with this technology.

Change the reader options by modifying the object and using the setter to change it:

```
opt = tech.save_layout_options
opt.dbu = 0.01
tech.save_layout_options = opt
```

Python specific notes:
The object exposes a readable attribute 'save_layout_options'. This is the getter.

save_layout_options=

Signature: void **save_layout_options=** (const SaveLayoutOptions options)

Description: Sets the layout writer options

See save_layout_options for a description of this property.

Python specific notes:
The object exposes a writable attribute 'save_layout_options'. This is the setter.

set_default_grids

Signature: void **set_default_grids** (double[] grids, double default_grid = 0)

Description: Sets the default grids and the strong default one

See default_grids and default_grid for a description of this property. Note that the default grid has to be a member of the 'grids' array to become active.

This method has been introduced in version 0.29.

technologies_from_xml

Signature: *[static]* void **technologies_from_xml** (string xml)

Description: Loads the technologies from a XML representation

See technologies_to_xml for details.

technologies_to_xml

Signature: *[static]* string **technologies_to_xml**

Description: Returns a XML representation of all technologies registered in the system

technologies_from_xml can be used to restore the technology definitions. This method is provided mainly as a substitute for the pre-0.25 way of accessing technology data through the 'technology-data' configuration parameter. This method will return the equivalent string.

technology_by_name

Signature: *[static]* Technology ptr **technology_by_name** (string name)

Description: Gets the technology object for a given name

technology_from_xml

Signature: *[static]* Technology **technology_from_xml** (string xml)

Description: Loads the technology from a XML representation

	See technology_to_xml for details. Note that this function will create a new Technology object which is not registered in the system. See Technology#register for details.
technology_names	**Signature**: *[static]* string[] **technology_names** **Description**: Gets a list of technology names defined in the system
to_xml	**Signature**: *[const]* string **to_xml** **Description**: Returns a XML representation of this technolog technology_from_xml can be used to restore the technology definition.

4.99. API reference - Class Text

Notation used in Ruby API documentation

Module: db

Description: A text object

A text object has a point (location), a text, a text transformation, a text size and a font id. Text size and font id are provided to be be able to render the text correctly. Text objects are used as labels (i.e. for pins) or to indicate a particular position.

The Text class uses integer coordinates. A class that operates with floating-point coordinates is DText.

See The Database API for more details about the database objects.

Public constructors

new Text ptr	**new**	(const DText dtext)	Creates an integer coordinate text from a floating-point coordinate text
new Text ptr	**new**		Default constructor
new Text ptr	**new**	(string string, const Trans trans)	Constructor with string and transformation
new Text ptr	**new**	(string string, int x, int y)	Constructor with string and location
new Text ptr	**new**	(string string, const Trans trans, int height, int font)	Constructor with string, transformation, text height and font

Public methods

[const]	bool	**!=**	(const Text text)	Inequality
[const]	bool	**<**	(const Text t)	Less operator
[const]	bool	**==**	(const Text text)	Equality
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

	void	**assign**	(const Text other)	Assigns another object to self
[const]	Box	**bbox**		Gets the bounding box of the text
[const]	new Text ptr	**dup**		Creates a copy of self
[const]	int	**font**		Gets the font number
	void	**font=**	(int f)	Sets the font number
[const]	HAlign	**halign**		Gets the horizontal alignment
	void	**halign=**	(HAlign a)	Sets the horizontal alignment
[const]	unsigned long	**hash**		Computes a hash value
	Text	**move**	(const Vector distance)	Moves the text by a certain distance (modifies self)
	Text	**move**	(int dx, int dy)	Moves the text by a certain distance (modifies self)
[const]	Text	**moved**	(const Vector distance)	Returns the text moved by a certain distance (does not modify self)
[const]	Text	**moved**	(int dx, int dy)	Returns the text moved by a certain distance (does not modify self)
[const]	Point	**position**		Gets the position of the text
[const]	int	**size**		Gets the text height
	void	**size=**	(int s)	Sets the text height of this object
[const]	string	**string**		Get the text string
	void	**string=**	(string text)	Assign a text string to this object
[const]	DText	**to_dtype**	(double dbu = 1)	Converts the text to a floating-point coordinate text
[const]	string	**to_s**	(double dbu = 0)	Converts the object to a string.
[const]	Trans	**trans**		Gets the transformation
	void	**trans=**	(const Trans t)	Assign a transformation (text position and orientation) to this object
[const]	Text	**transformed**	(const ICplxTrans t)	Transform the text with the given complex transformation
[const]	Text	**transformed**	(const Trans t)	Transforms the text with the given simple transformation
[const]	DText	**transformed**	(const CplxTrans t)	Transforms the text with the given complex transformation

[const]	VAlign	**valign**		Gets the vertical alignment
	void	**valign=**	(VAlign a)	Sets the vertical alignment
[const]	int	**x**		Gets the x location of the text
	void	**x=**	(int x)	Sets the x location of the text
[const]	int	**y**		Gets the y location of the text
	void	**y=**	(int y)	Sets the y location of the text

Public static methods and constants

[static,const]	HAlign	**HAlignCenter**		Centered horizontal alignment
[static,const]	HAlign	**HAlignLeft**		Left horizontal alignment
[static,const]	HAlign	**HAlignRight**		Right horizontal alignment
[static,const]	HAlign	**NoHAlign**		Undefined horizontal alignment
[static,const]	VAlign	**NoVAlign**		Undefined vertical alignment
[static,const]	VAlign	**VAlignBottom**		Bottom vertical alignment
[static,const]	VAlign	**VAlignCenter**		Centered vertical alignment
[static,const]	VAlign	**VAlignTop**		Top vertical alignment
	new Text ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
	void	**halign=**	(int a)	Use of this method is deprecated
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
	void	**valign=**	(int a)	Use of this method is deprecated

Detailed description

!=

Signature: *[const]* bool **!=** (const Text text)
Description: Inequality

	Return true, if this text object and the given text are not equal
<	**Signature**: *[const]* bool **<** (const Text t) **Description**: Less operator **t**: The object to compare against This operator is provided to establish some, not necessarily a certain sorting order
==	**Signature**: *[const]* bool **==** (const Text text) **Description**: Equality Return true, if this text object and the given text are equal
HAlignCenter	**Signature**: *[static,const]* HAlign **HAlignCenter** **Description**: Centered horizontal alignment **Python specific notes:** The object exposes a readable attribute 'HAlignCenter'. This is the getter.
HAlignLeft	**Signature**: *[static,const]* HAlign **HAlignLeft** **Description**: Left horizontal alignment **Python specific notes:** The object exposes a readable attribute 'HAlignLeft'. This is the getter.
HAlignRight	**Signature**: *[static,const]* HAlign **HAlignRight** **Description**: Right horizontal alignment **Python specific notes:** The object exposes a readable attribute 'HAlignRight'. This is the getter.
NoHAlign	**Signature**: *[static,const]* HAlign **NoHAlign** **Description**: Undefined horizontal alignment **Python specific notes:** The object exposes a readable attribute 'NoHAlign'. This is the getter.
NoVAlign	**Signature**: *[static,const]* VAlign **NoVAlign** **Description**: Undefined vertical alignment **Python specific notes:** The object exposes a readable attribute 'NoVAlign'. This is the getter.
VAlignBottom	**Signature**: *[static,const]* VAlign **VAlignBottom** **Description**: Bottom vertical alignment **Python specific notes:** The object exposes a readable attribute 'VAlignBottom'. This is the getter.
VAlignCenter	**Signature**: *[static,const]* VAlign **VAlignCenter** **Description**: Centered vertical alignment **Python specific notes:** The object exposes a readable attribute 'VAlignCenter'. This is the getter.

VAlignTop

Signature: *[static,const]* VAlign **VAlignTop**

Description: Top vertical alignment

Python specific notes:
The object exposes a readable attribute 'VAlignTop'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Text other)

Description: Assigns another object to self

bbox

Signature: *[const]* Box **bbox**

Description: Gets the bounding box of the text

The bounding box of the text is a single point - the location of the text. Both points of the box are identical.

This method has been added in version 0.28.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Text ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

font

Signature: *[const]* int **font**

Description: Gets the font number

See font= for a description of this property.

Python specific notes:
The object exposes a readable attribute 'font'. This is the getter.

font=

Signature: void **font=** (int f)

Description: Sets the font number

The font number does not play a role for KLayout. This property is provided for compatibility with other systems which allow using different fonts for the text objects.

Python specific notes:
The object exposes a writable attribute 'font'. This is the setter.

from_s

Signature: *[static]* new Text ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

halign

Signature: *[const]* HAlign **halign**

Description: Gets the horizontal alignment

See halign= for a description of this property.

Python specific notes:
The object exposes a readable attribute 'halign'. This is the getter.

halign=

(1) Signature: void **halign=** (int a)

Description: Sets the horizontal alignment

Use of this method is deprecated

This is the version accepting integer values. It's provided for backward compatibility.

Python specific notes:
The object exposes a writable attribute 'halign'. This is the setter.

(2) Signature: void **halign=** (HAlign a)

Description: Sets the horizontal alignment

This property specifies how the text is aligned relative to the anchor point. This property has been introduced in version 0.22 and extended to enums in 0.28.

Python specific notes:
The object exposes a writable attribute 'halign'. This is the setter.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given text object. This method enables texts as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

move

(1) Signature: Text **move** (const Vector distance)

Description: Moves the text by a certain distance (modifies self)

p:	The offset to move the text.
Returns:	A reference to this text object

Moves the text by a given offset and returns the moved text. Does not check for coordinate overflows.

(2) Signature: Text **move** (int dx, int dy)

Description: Moves the text by a certain distance (modifies self)

dx: The x distance to move the text.

dy: The y distance to move the text.

Returns: A reference to this text object

Moves the text by a given distance in x and y direction and returns the moved text. Does not check for coordinate overflows.

This method was introduced in version 0.23.

moved

(1) Signature: *[const]* Text **moved** (const Vector distance)

Description: Returns the text moved by a certain distance (does not modify self)

p: The offset to move the text.

Returns: The moved text.

Moves the text by a given offset and returns the moved text. Does not modify *this. Does not check for coordinate overflows.

(2) Signature: *[const]* Text **moved** (int dx, int dy)

Description: Returns the text moved by a certain distance (does not modify self)

dx: The x distance to move the text.

dy: The y distance to move the text.

Returns: The moved text.

Moves the text by a given offset and returns the moved text. Does not modify *this. Does not check for coordinate overflows.

This method was introduced in version 0.23.

new

(1) Signature: *[static]* new Text ptr **new** (const DText dtext)

Description: Creates an integer coordinate text from a floating-point coordinate text

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dtext'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Text ptr **new**

Description: Default constructor

Creates a text with unit transformation and empty text.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Text ptr **new** (string string, const Trans trans)

Description: Constructor with string and transformation

A string and a transformation is provided to this constructor. The transformation specifies the location and orientation of the text object.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Text ptr **new** (string string, int x, int y)

Description: Constructor with string and location

A string and a location is provided to this constructor. The location is specifies as a pair of x and y coordinates.

This method has been introduced in version 0.23.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new Text ptr **new** (string string, const Trans trans, int height, int font)

Description: Constructor with string, transformation, text height and font

A string and a transformation is provided to this constructor. The transformation specifies the location and orientation of the text object. In addition, the text height and font can be specified.

Python specific notes:
This method is the default initializer of the object.

position

Signature: *[const]* Point **position**

Description: Gets the position of the text

This convenience method has been added in version 0.28.

size

Signature: *[const]* int **size**

Description: Gets the text height

Python specific notes:
The object exposes a readable attribute 'size'. This is the getter.

size=

Signature: void **size=** (int s)

Description: Sets the text height of this object

Python specific notes:
The object exposes a writable attribute 'size'. This is the setter.

string

Signature: *[const]* string **string**

Description: Get the text string

Python specific notes:
The object exposes a readable attribute 'string'. This is the getter.

string=

Signature: void **string=** (string text)

Description: Assign a text string to this object

Python specific notes:
The object exposes a writable attribute 'string'. This is the setter.

to_dtype

Signature: *[const]* DText **to_dtype** (double dbu = 1)

Description: Converts the text to a floating-point coordinate text

The database unit can be specified to translate the integer-coordinate text into a floating-point coordinate text in micron units. The database unit is basically a scaling factor.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: Converts the object to a string.

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

trans

Signature: *[const]* Trans **trans**

Description: Gets the transformation

Python specific notes:
The object exposes a readable attribute 'trans'. This is the getter.

trans=

Signature: void **trans=** (const Trans t)

Description: Assign a transformation (text position and orientation) to this object

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.

transformed

(1) Signature: *[const]* Text **transformed** (const ICplxTrans t)

Description: Transform the text with the given complex transformation

t:	The magnifying transformation to apply
Returns:	The transformed text (in this case an integer coordinate object now)

This method has been introduced in version 0.18.

(2) Signature: *[const]* Text **transformed** (const Trans t)

Description: Transforms the text with the given simple transformation

t:	The transformation to apply
Returns:	The transformed text

(3) Signature: *[const]* DText **transformed** (const CplxTrans t)

Description: Transforms the text with the given complex transformation

t:	The magnifying transformation to apply
Returns:	The transformed text (a DText now)

valign

Signature: *[const]* VAlign **valign**

Description: Gets the vertical alignment

See valign= for a description of this property.

Python specific notes:
The object exposes a readable attribute 'valign'. This is the getter.

valign=

(1) Signature: void **valign=** (int a)

Description: Sets the vertical alignment

Use of this method is deprecated

This is the version accepting integer values. It's provided for backward compatibility.

Python specific notes:

	The object exposes a writable attribute 'valign'. This is the setter. **(2) Signature**: void **valign=** (VAlign a) **Description**: Sets the vertical alignment This property specifies how the text is aligned relative to the anchor point. This property has been introduced in version 0.22 and extended to enums in 0.28. **Python specific notes:** The object exposes a writable attribute 'valign'. This is the setter.
x	**Signature**: *[const]* int **x** **Description**: Gets the x location of the text This method has been introduced in version 0.23. **Python specific notes:** The object exposes a readable attribute 'x'. This is the getter.
x=	**Signature**: void **x=** (int x) **Description**: Sets the x location of the text This method has been introduced in version 0.23. **Python specific notes:** The object exposes a writable attribute 'x'. This is the setter.
y	**Signature**: *[const]* int **y** **Description**: Gets the y location of the text This method has been introduced in version 0.23. **Python specific notes:** The object exposes a readable attribute 'y'. This is the getter.
y=	**Signature**: void **y=** (int y) **Description**: Sets the y location of the text This method has been introduced in version 0.23. **Python specific notes:** The object exposes a writable attribute 'y'. This is the setter.

4.100. API reference - Class DText

Notation used in Ruby API documentation

Module: db

Description: A text object

A text object has a point (location), a text, a text transformation, a text size and a font id. Text size and font id are provided to be be able to render the text correctly. Text objects are used as labels (i.e. for pins) or to indicate a particular position.

The DText class uses floating-point coordinates. A class that operates with integer coordinates is Text.

See The Database API for more details about the database objects.

Public constructors

new DText ptr	**new**	(const Text Text)	Creates a floating-point coordinate text from an integer coordinate text
new DText ptr	**new**		Default constructor
new DText ptr	**new**	(string string, const DTrans trans)	Constructor with string and transformation
new DText ptr	**new**	(string string, double x, double y)	Constructor with string and location
new DText ptr	**new**	(string string, const DTrans trans, double height, int font)	Constructor with string, transformation, text height and font

Public methods

[const]	bool	**!=**	(const DText text)	Inequality
[const]	bool	**<**	(const DText t)	Less operator
[const]	bool	**==**	(const DText text)	Equality
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

	void	**assign**	(const DText other)	Assigns another object to self
[const]	DBox	**bbox**		Gets the bounding box of the text
[const]	new DText ptr	**dup**		Creates a copy of self
[const]	int	**font**		Gets the font number
	void	**font=**	(int f)	Sets the font number
[const]	HAlign	**halign**		Gets the horizontal alignment
	void	**halign=**	(HAlign a)	Sets the horizontal alignment
[const]	unsigned long	**hash**		Computes a hash value
	DText	**move**	(const DVector distance)	Moves the text by a certain distance (modifies self)
	DText	**move**	(double dx, double dy)	Moves the text by a certain distance (modifies self)
[const]	DText	**moved**	(const DVector distance)	Returns the text moved by a certain distance (does not modify self)
[const]	DText	**moved**	(double dx, double dy)	Returns the text moved by a certain distance (does not modify self)
[const]	DPoint	**position**		Gets the position of the text
[const]	double	**size**		Gets the text height
	void	**size=**	(double s)	Sets the text height of this object
[const]	string	**string**		Get the text string
	void	**string=**	(string text)	Assign a text string to this object
[const]	Text	**to_itype**	(double dbu = 1)	Converts the text to an integer coordinate text
[const]	string	**to_s**	(double dbu = 0)	Converts the object to a string.
[const]	DTrans	**trans**		Gets the transformation
	void	**trans=**	(const DTrans t)	Assign a transformation (text position and orientation) to this object
[const]	Text	**transformed**	(const VCplxTrans t)	Transforms the text with the given complex transformation
[const]	DText	**transformed**	(const DTrans t)	Transforms the text with the given simple transformation
[const]	DText	**transformed**	(const DCplxTrans t)	Transforms the text with the given complex transformation
[const]	VAlign	**valign**		Gets the vertical alignment

	void	valign=	(VAlign a)	Sets the vertical alignment
[const]	double	x		Gets the x location of the text
	void	x=	(double x)	Sets the x location of the text
[const]	double	y		Gets the y location of the text
	void	y=	(double y)	Sets the y location of the text

Public static methods and constants

[static,const]	HAlign	HAlignCenter		Centered horizontal alignment
[static,const]	HAlign	HAlignLeft		Left horizontal alignment
[static,const]	HAlign	HAlignRight		Right horizontal alignment
[static,const]	HAlign	NoHAlign		Undefined horizontal alignment
[static,const]	VAlign	NoVAlign		Undefined vertical alignment
[static,const]	VAlign	VAlignBottom		Bottom vertical alignment
[static,const]	VAlign	VAlignCenter		Centered vertical alignment
[static,const]	VAlign	VAlignTop		Top vertical alignment
	new DText ptr	from_s	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	create		Use of this method is deprecated. Use _create instead
	void	destroy		Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?		Use of this method is deprecated. Use _destroyed? instead
	void	halign=	(int a)	Use of this method is deprecated
[const]	bool	is_const_object?		Use of this method is deprecated. Use _is_const_object? instead
	void	valign=	(int a)	Use of this method is deprecated

Detailed description

!=

Signature: *[const]* bool **!=** (const DText text)
Description: Inequality

Return true, if this text object and the given text are not equal

<	**Signature**: *[const]* bool **<** (const DText t) **Description**: Less operator **t**: The object to compare against This operator is provided to establish some, not necessarily a certain sorting order
==	**Signature**: *[const]* bool **==** (const DText text) **Description**: Equality Return true, if this text object and the given text are equal
HAlignCenter	**Signature**: *[static,const]* HAlign **HAlignCenter** **Description**: Centered horizontal alignment **Python specific notes:** The object exposes a readable attribute 'HAlignCenter'. This is the getter.
HAlignLeft	**Signature**: *[static,const]* HAlign **HAlignLeft** **Description**: Left horizontal alignment **Python specific notes:** The object exposes a readable attribute 'HAlignLeft'. This is the getter.
HAlignRight	**Signature**: *[static,const]* HAlign **HAlignRight** **Description**: Right horizontal alignment **Python specific notes:** The object exposes a readable attribute 'HAlignRight'. This is the getter.
NoHAlign	**Signature**: *[static,const]* HAlign **NoHAlign** **Description**: Undefined horizontal alignment **Python specific notes:** The object exposes a readable attribute 'NoHAlign'. This is the getter.
NoVAlign	**Signature**: *[static,const]* VAlign **NoVAlign** **Description**: Undefined vertical alignment **Python specific notes:** The object exposes a readable attribute 'NoVAlign'. This is the getter.
VAlignBottom	**Signature**: *[static,const]* VAlign **VAlignBottom** **Description**: Bottom vertical alignment **Python specific notes:** The object exposes a readable attribute 'VAlignBottom'. This is the getter.
VAlignCenter	**Signature**: *[static,const]* VAlign **VAlignCenter** **Description**: Centered vertical alignment **Python specific notes:** The object exposes a readable attribute 'VAlignCenter'. This is the getter.

VAlignTop

Signature: *[static,const]* VAlign **VAlignTop**

Description: Top vertical alignment

Python specific notes:
The object exposes a readable attribute 'VAlignTop'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const DText other)

Description: Assigns another object to self

bbox

Signature: *[const]* DBox **bbox**

Description: Gets the bounding box of the text

The bounding box of the text is a single point - the location of the text. Both points of the box are identical.

This method has been added in version 0.28.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new DText ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

font

Signature: *[const]* int **font**

Description: Gets the font number

See font= for a description of this property.

Python specific notes:
The object exposes a readable attribute 'font'. This is the getter.

font=

Signature: void **font=** (int f)

Description: Sets the font number

The font number does not play a role for KLayout. This property is provided for compatibility with other systems which allow using different fonts for the text objects.

Python specific notes:
The object exposes a writable attribute 'font'. This is the setter.

from_s

Signature: *[static]* new DText ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

halign

Signature: *[const]* HAlign **halign**

Description: Gets the horizontal alignment

See halign= for a description of this property.

Python specific notes:
The object exposes a readable attribute 'halign'. This is the getter.

halign=

(1) Signature: void **halign=** (int a)

Description: Sets the horizontal alignment

Use of this method is deprecated

This is the version accepting integer values. It's provided for backward compatibility.

Python specific notes:
The object exposes a writable attribute 'halign'. This is the setter.

(2) Signature: void **halign=** (HAlign a)

Description: Sets the horizontal alignment

This property specifies how the text is aligned relative to the anchor point. This property has been introduced in version 0.22 and extended to enums in 0.28.

Python specific notes:
The object exposes a writable attribute 'halign'. This is the setter.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given text object. This method enables texts as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

move

(1) Signature: DText **move** (const DVector distance)

Description: Moves the text by a certain distance (modifies self)

p:	The offset to move the text.
Returns:	A reference to this text object

Moves the text by a given offset and returns the moved text. Does not check for coordinate overflows.

(2) Signature: DText **move** (double dx, double dy)

Description: Moves the text by a certain distance (modifies self)

dx:	The x distance to move the text.
dy:	The y distance to move the text.
Returns:	A reference to this text object

Moves the text by a given distance in x and y direction and returns the moved text. Does not check for coordinate overflows.

This method was introduced in version 0.23.

moved

(1) Signature: *[const]* DText **moved** (const DVector distance)

Description: Returns the text moved by a certain distance (does not modify self)

p:	The offset to move the text.
Returns:	The moved text.

Moves the text by a given offset and returns the moved text. Does not modify *this. Does not check for coordinate overflows.

(2) Signature: *[const]* DText **moved** (double dx, double dy)

Description: Returns the text moved by a certain distance (does not modify self)

dx:	The x distance to move the text.
dy:	The y distance to move the text.
Returns:	The moved text.

Moves the text by a given offset and returns the moved text. Does not modify *this. Does not check for coordinate overflows.

This method was introduced in version 0.23.

new

(1) Signature: *[static]* new DText ptr **new** (const Text Text)

Description: Creates a floating-point coordinate text from an integer coordinate text

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_itext'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DText ptr **new**

Description: Default constructor

Creates a text with unit transformation and empty text.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new DText ptr **new** (string string, const DTrans trans)

Description: Constructor with string and transformation

A string and a transformation is provided to this constructor. The transformation specifies the location and orientation of the text object.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new DText ptr **new** (string string, double x, double y)

Description: Constructor with string and location

A string and a location is provided to this constructor. The location is specifies as a pair of x and y coordinates.

This method has been introduced in version 0.23.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new DText ptr **new** (string string, const DTrans trans, double height, int font)

Description: Constructor with string, transformation, text height and font

A string and a transformation is provided to this constructor. The transformation specifies the location and orientation of the text object. In addition, the text height and font can be specified.

Python specific notes:
This method is the default initializer of the object.

position

Signature: *[const]* DPoint **position**

Description: Gets the position of the text

This convenience method has been added in version 0.28.

size

Signature: *[const]* double **size**

Description: Gets the text height

Python specific notes:
The object exposes a readable attribute 'size'. This is the getter.

size=

Signature: void **size=** (double s)

Description: Sets the text height of this object

Python specific notes:
The object exposes a writable attribute 'size'. This is the setter.

string

Signature: *[const]* string **string**

Description: Get the text string

Python specific notes:
The object exposes a readable attribute 'string'. This is the getter.

string=

Signature: void **string=** (string text)

Description: Assign a text string to this object

Python specific notes:
The object exposes a writable attribute 'string'. This is the setter.

to_itype

Signature: *[const]* Text **to_itype** (double dbu = 1)

Description: Converts the text to an integer coordinate text

The database unit can be specified to translate the floating-point coordinate Text in micron units to an integer-coordinate text in database units. The text's coordinates will be divided by the database unit.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: Converts the object to a string.

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

trans

Signature: *[const]* DTrans **trans**

Description: Gets the transformation

Python specific notes:
The object exposes a readable attribute 'trans'. This is the getter.

trans=

Signature: void **trans=** (const DTrans t)

Description: Assign a transformation (text position and orientation) to this object

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.

transformed

(1) Signature: *[const]* Text **transformed** (const VCplxTrans t)

Description: Transforms the text with the given complex transformation

t:	The magnifying transformation to apply
Returns:	The transformed text (in this case an integer coordinate text)

This method has been introduced in version 0.25.

(2) Signature: *[const]* DText **transformed** (const DTrans t)

Description: Transforms the text with the given simple transformation

t:	The transformation to apply
Returns:	The transformed text

(3) Signature: *[const]* DText **transformed** (const DCplxTrans t)

Description: Transforms the text with the given complex transformation

t:	The magnifying transformation to apply
Returns:	The transformed text (a DText now)

valign

Signature: *[const]* VAlign **valign**

Description: Gets the vertical alignment

See valign= for a description of this property.

Python specific notes:
The object exposes a readable attribute 'valign'. This is the getter.

valign=

(1) Signature: void **valign=** (int a)

Description: Sets the vertical alignment

Use of this method is deprecated

This is the version accepting integer values. It's provided for backward compatibility.

Python specific notes:
The object exposes a writable attribute 'valign'. This is the setter.

(2) Signature: void **valign=** (VAlign a)

Description: Sets the vertical alignment

This property specifies how the text is aligned relative to the anchor point. This property has been introduced in version 0.22 and extended to enums in 0.28.

Python specific notes:
The object exposes a writable attribute 'valign'. This is the setter.

x

Signature: *[const]* double **x**

Description: Gets the x location of the text

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'x'. This is the getter.

x=

Signature: void **x=** (double x)

Description: Sets the x location of the text

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'x'. This is the setter.

y

Signature: *[const]* double **y**

Description: Gets the y location of the text

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'y'. This is the getter.

y=

Signature: void **y=** (double y)

Description: Sets the y location of the text

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'y'. This is the setter.

4.101. API reference - Class HAlign

Notation used in Ruby API documentation

Module: db

Description: This class represents the horizontal alignment modes.

This enum has been introduced in version 0.28.

Public constructors

new HAlign ptr	**new**	(int i)		Creates an enum from an integer value
new HAlign ptr	**new**	(string s)		Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const HAlign other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const HAlign other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const HAlign other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const HAlign other)	Assigns another object to self
[const]	new HAlign ptr	**dup**		Creates a copy of self
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string

[const]	int	**to_i**	Gets the integer value from the enum
[const]	string	**to_s**	Gets the symbolic string from an enum

Public static methods and constants

[static,const]	HAlign	**HAlignCenter**	Centered horizontal alignment
[static,const]	HAlign	**HAlignLeft**	Left horizontal alignment
[static,const]	HAlign	**HAlignRight**	Right horizontal alignment
[static,const]	HAlign	**NoHAlign**	Undefined horizontal alignment

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

(1) Signature: *[const]* bool **!=** (const HAlign other)
Description: Compares two enums for inequality

(2) Signature: *[const]* bool **!=** (int other)
Description: Compares an enum with an integer for inequality

<

(1) Signature: *[const]* bool **<** (const HAlign other)
Description: Returns true if the first enum is less (in the enum symbol order) than the second

(2) Signature: *[const]* bool **<** (int other)
Description: Returns true if the enum is less (in the enum symbol order) than the integer value

==

(1) Signature: *[const]* bool **==** (const HAlign other)
Description: Compares two enums

(2) Signature: *[const]* bool **==** (int other)
Description: Compares an enum with an integer value

HAlignCenter

Signature: *[static,const]* HAlign **HAlignCenter**
Description: Centered horizontal alignment
Python specific notes:

The object exposes a readable attribute 'HAlignCenter'. This is the getter.

HAlignLeft

Signature: *[static,const]* HAlign **HAlignLeft**

Description: Left horizontal alignment

Python specific notes:
The object exposes a readable attribute 'HAlignLeft'. This is the getter.

HAlignRight

Signature: *[static,const]* HAlign **HAlignRight**

Description: Right horizontal alignment

Python specific notes:
The object exposes a readable attribute 'HAlignRight'. This is the getter.

NoHAlign

Signature: *[static,const]* HAlign **NoHAlign**

Description: Undefined horizontal alignment

Python specific notes:
The object exposes a readable attribute 'NoHAlign'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const HAlign other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new HAlign ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:

	This method is also available as 'repr(object)'.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**(1) Signature**: *[static]* new HAlign ptr **new** (int i) **Description**: Creates an enum from an integer value **Python specific notes:** This method is the default initializer of the object. **(2) Signature**: *[static]* new HAlign ptr **new** (string s) **Description**: Creates an enum from a string value **Python specific notes:** This method is the default initializer of the object.
to_i	**Signature**: *[const]* int **to_i** **Description**: Gets the integer value from the enum **Python specific notes:** This method is also available as 'int(object)'.
to_s	**Signature**: *[const]* string **to_s** **Description**: Gets the symbolic string from an enum **Python specific notes:** This method is also available as 'str(object)'.

4.102. API reference - Class VAlign

Notation used in Ruby API documentation

Module: db

Description: This class represents the vertical alignment modes.

This enum has been introduced in version 0.28.

Public constructors

new VAlign ptr	**new**	(int i)		Creates an enum from an integer value
new VAlign ptr	**new**	(string s)		Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const VAlign other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const VAlign other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const VAlign other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const VAlign other)	Assigns another object to self
[const]	new VAlign ptr	**dup**		Creates a copy of self
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string

[const]	int	**to_i**	Gets the integer value from the enum
[const]	string	**to_s**	Gets the symbolic string from an enum

Public static methods and constants

[static,const]	VAlign	**NoVAlign**	Undefined vertical alignment
[static,const]	VAlign	**VAlignBottom**	Bottom vertical alignment
[static,const]	VAlign	**VAlignCenter**	Centered vertical alignment
[static,const]	VAlign	**VAlignTop**	Top vertical alignment

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

(1) Signature: *[const]* bool **!=** (const VAlign other)
Description: Compares two enums for inequality

(2) Signature: *[const]* bool **!=** (int other)
Description: Compares an enum with an integer for inequality

<

(1) Signature: *[const]* bool **<** (const VAlign other)
Description: Returns true if the first enum is less (in the enum symbol order) than the second

(2) Signature: *[const]* bool **<** (int other)
Description: Returns true if the enum is less (in the enum symbol order) than the integer value

==

(1) Signature: *[const]* bool **==** (const VAlign other)
Description: Compares two enums

(2) Signature: *[const]* bool **==** (int other)
Description: Compares an enum with an integer value

NoVAlign

Signature: *[static,const]* VAlign **NoVAlign**
Description: Undefined vertical alignment
Python specific notes:

The object exposes a readable attribute 'NoVAlign'. This is the getter.

VAlignBottom

Signature: *[static,const]* VAlign **VAlignBottom**

Description: Bottom vertical alignment

Python specific notes:
The object exposes a readable attribute 'VAlignBottom'. This is the getter.

VAlignCenter

Signature: *[static,const]* VAlign **VAlignCenter**

Description: Centered vertical alignment

Python specific notes:
The object exposes a readable attribute 'VAlignCenter'. This is the getter.

VAlignTop

Signature: *[static,const]* VAlign **VAlignTop**

Description: Top vertical alignment

Python specific notes:
The object exposes a readable attribute 'VAlignTop'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const VAlign other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new VAlign ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:

	This method is also available as 'repr(object)'.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**(1) Signature**: *[static]* new VAlign ptr **new** (int i) **Description**: Creates an enum from an integer value **Python specific notes:** This method is the default initializer of the object. **(2) Signature**: *[static]* new VAlign ptr **new** (string s) **Description**: Creates an enum from a string value **Python specific notes:** This method is the default initializer of the object.
to_i	**Signature**: *[const]* int **to_i** **Description**: Gets the integer value from the enum **Python specific notes:** This method is also available as 'int(object)'.
to_s	**Signature**: *[const]* string **to_s** **Description**: Gets the symbolic string from an enum **Python specific notes:** This method is also available as 'str(object)'.

4.103. API reference - Class TileOutputReceiver

Notation used in Ruby API documentation

Module: db

Description: A receiver abstraction for the tiling processor.

Class hierarchy: TileOutputReceiver

The tiling processor (TilingProcessor) is a framework for executing sequences of operations on tiles of a layout or multiple layouts. The TileOutputReceiver class is used to specify an output channel for the tiling processor. See TilingProcessor#output for more details.

This class has been introduced in version 0.23.

Public constructors

new TileOutputReceiver ptr	**new**		Creates a new object of this class

Public methods

	void	**_assign**	(const TileOutputReceiver other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new TileOutputReceiver ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const TileOutputReceiver other)	Assigns another object to self
[virtual]	void	**begin**	(unsigned long nx, unsigned long ny, const DPoint p0, double dx, double dy, const DBox frame)	Initiates the delivery
[const]	new TileOutputReceiver ptr	**dup**		Creates a copy of self

[virtual]	void	**finish**	(bool success)	Indicates the end of the execution
[const]	TilingProcessor ptr	**processor**		Gets the processor the receiver is attached to
[virtual]	void	**put**	(unsigned long ix, unsigned long iy, const Box tile, variant obj, double dbu, bool clip)	Delivers data for one tile

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_assign

Signature: void **_assign** (const TileOutputReceiver other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new TileOutputReceiver ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const TileOutputReceiver other)

Description: Assigns another object to self

begin

Signature: *[virtual]* void **begin** (unsigned long nx, unsigned long ny, const DPoint p0, double dx, double dy, const DBox frame)

Description: Initiates the delivery

nx:	The number of tiles in x direction
ny:	The number of tiles in y direction
p0:	The initial point
dx:	The tile's x dimension
dy:	The tile's y dimension
frame:	The overall frame that is the basis of the tiling

This method is called before the first tile delivers its data.

The tile's coordinates will be p0+(ix*dx,iy*dy)..p0+((ix+1)*dx,(iy+1)*dy) where ix=0..nx-1, iy=0..ny-1.

All coordinates are given in micron. If tiles are not used, nx and ny are 0.

The frame parameter has been added in version 0.25.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new TileOutputReceiver ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

finish

Signature: *[virtual]* void **finish** (bool success)

Description: Indicates the end of the execution

This method is called when the tiling processor has finished the last tile and script item. The success flag is set to true, if every tile has finished successfully. Otherwise, this value is false.

The success flag has been added in version 0.25.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new TileOutputReceiver ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

processor

Signature: *[const]* TilingProcessor ptr **processor**

Description: Gets the processor the receiver is attached to

This attribute is set before begin and can be nil if the receiver is not attached to a processor.

This method has been introduced in version 0.25.

put

Signature: *[virtual]* void **put** (unsigned long ix, unsigned long iy, const Box tile, variant obj, double dbu, bool clip)

Description: Delivers data for one tile

ix:	The x index of the tile
iy:	The y index of the tile

tile:	The tile's box
obj:	The object which is delivered
dbu:	The database unit
clip:	True if clipping at the tile box is requested

When the script's "_output" function is called, the data will be delivered through this method. "obj" is the data passed as the second argument to _output. The interpretation of the object remains subject to the implementation.

The obj and clip parameters are taken from the _output method call inside the script. If clip is set to true, this usually means that output shall be clipped to the tile.

4.104. API reference - Class TilingProcessor

Notation used in Ruby API documentation

Module: db

Description: A processor for layout which distributes tasks over tiles

The tiling processor executes one or several scripts on one or multiple layouts providing a tiling scheme. In that scheme, the processor divides the original layout into rectangular tiles and executes the scripts on each tile separately. The tiling processor allows one to specify multiple, independent scripts which are run separately on each tile. It can make use of multi-core CPU's by supporting multiple threads running the tasks in parallel (with respect to tiles and scripts).

Tiling a optional - if no tiles are specified, the tiling processing basically operates flat and parallelization extends to the scripts only.

Tiles can be overlapping to gather input from neighboring tiles into the current tile. In order to provide that feature, a border can be specified which gives the amount by which the search region is extended beyond the border of the tile. To specify the border, use the TilingProcessor#tile_border method.

The basis of the tiling processor are Region objects and expressions. Expressions are a built-in simple language to form simple scripts. Expressions allow access to the objects and methods built into KLayout. Each script can consist of multiple operations. Scripts are specified using TilingProcessor#queue.

Input is provided to the script through variables holding a Region object each. From outside the tiling processor, input is specified with the TilingProcessor#input method. This method is given a name and a RecursiveShapeIterator object which delivers the data for the input. On the script side, a Region object is provided through a variable named like the first argument of the "input" method.

Inside the script the following functions are provided:

- "_dbu" delivers the database unit used for the computations
- "_tile" delivers a region containing a mask for the tile (a rectangle) or nil if no tiling is used
- "_output" is used to deliver output (see below)

Output can be obtained from the tiling processor by registering a receiver with a channel. A channel is basically a name. Inside the script, the name describes a variable which can be used as the first argument of the "_output" function to identify the channel. A channel is registers using the TilingProcessor#output method. Beside the name, a receiver must be specified. A receiver is either another layout (a cell of that), a report database or a custom receiver implemented through the TileOutputReceiver class.

The "_output" function expects two or three parameters: one channel id (the variable that was defined by the name given in the output method call) and an object to output (a Region, Edges, EdgePairs or a geometrical primitive such as Polygon or Box). In addition, a boolean argument can be given indicating whether clipping at the tile shall be applied. If clipping is requested (the default), the shapes will be clipped at the tile's box.

The tiling can be specified either through a tile size, a tile number or both. If a tile size is specified with the TilingProcessor#tile_size method, the tiling processor will compute the number of tiles required. If the tile count is given (through TilingProcessor#tiles), the tile size will be computed. If both are given, the tiling array is fixed and the array is centered around the original layout's center. If the tiling origin is given as well, the tiling processor will use the given array without any modifications.

Once the tiling processor has been set up, the operation can be launched using TilingProcessor#execute.

This is some sample code. It performs two XOR operations between two layouts and delivers the results to a report database:

```
ly1 = ... # first layout
ly2 = ... # second layout

rdb = RBA::ReportDatabase::new("xor")
output_cell = rdb.create_cell(ly1.top_cell.name)
output_cat1 = rbd.create_category("XOR 1-10")
output_cat2 = rbd.create_category("XOR 2-11")

tp = RBA::TilingProcessor::new
tp.input("a1", ly1, ly1.top_cell.cell_index, RBA::LayerInfo::new(1, 0))
tp.input("a2", ly1, ly1.top_cell.cell_index, RBA::LayerInfo::new(2, 0))
tp.input("b1", ly2, ly2.top_cell.cell_index, RBA::LayerInfo::new(11, 0))
tp.input("b2", ly2, ly2.top_cell.cell_index, RBA::LayerInfo::new(12, 0))
```

```
tp.output("o1", rdb, output_cell, output_cat1)
tp.output("o2", rdb, output_cell, output_cat2)
tp.queue("_output(o1, a1 ^ b1)")
tp.queue("_output(o2, a2 ^ b2)")
tp.tile_size(50.0, 50.0)
tp.execute("Job description")
```

This class has been introduced in version 0.23.

Public constructors

new TilingProcessor ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const TilingProcessor other)	Assigns another object to self
[const]	double	**dbu**		Gets the database unit under which the computations will be done
	void	**dbu=**	(double u)	Sets the database unit under which the computations will be done
[const]	new TilingProcessor ptr	**dup**		Creates a copy of self
	void	**execute**	(string desc)	Runs the job
	void	**frame=**	(const DBox frame)	Sets the layout frame
	void	**input**	(string name, const RecursiveShapeIterator iter)	Specifies input for the tiling processor
	void	**input**	(string name, const RecursiveShapeIterator iter, const ICplxTrans trans)	Specifies input for the tiling processor
	void	**input**	(string name, const Layout layout,	Specifies input for the tiling processor

		unsigned int cell_index, const LayerInfo lp)	
void	**input**	(string name, const Layout layout, unsigned int cell_index, unsigned int layer)	Specifies input for the tiling processor
void	**input**	(string name, const Layout layout, unsigned int cell_index, const LayerInfo lp, const ICplxTrans trans)	Specifies input for the tiling processor
void	**input**	(string name, const Layout layout, unsigned int cell_index, unsigned int layer, const ICplxTrans trans)	Specifies input for the tiling processor
void	**input**	(string name, const Region region)	Specifies input for the tiling processor
void	**input**	(string name, const Region region, const ICplxTrans trans)	Specifies input for the tiling processor
void	**input**	(string name, const Edges edges)	Specifies input for the tiling processor
void	**input**	(string name, const Edges edges, const ICplxTrans trans)	Specifies input for the tiling processor
void	**input**	(string name, const EdgePairs edge_pairs)	Specifies input for the tiling processor
void	**input**	(string name, const EdgePairs edge_pairs, const ICplxTrans trans)	Specifies input for the tiling processor
void	**input**	(string name, const Texts texts)	Specifies input for the tiling processor
void	**input**	(string name, const Texts texts, const ICplxTrans trans)	Specifies input for the tiling processor
void	**output**	(string name, TileOutputReceiver ptr rec)	Specifies output for the tiling processor
void	**output**	(string name, Layout layout, unsigned int cell, const LayerInfo lp)	Specifies output to a layout layer
void	**output**	(string name, Layout layout, unsigned int cell,	Specifies output to a layout layer

			unsigned int layer_index)	
	void	**output**	(string name, Region region)	Specifies output to a Region object
	void	**output**	(string name, Edges edges)	Specifies output to an Edges object
	void	**output**	(string name, EdgePairs edge_pairs)	Specifies output to an EdgePairs object
	void	**output**	(string name, Texts texts)	Specifies output to an Texts object
	void	**output**	(string name, double ptr sum)	Specifies output to single value
	void	**output**	(string name, ReportDatabase rdb, unsigned long cell_id, unsigned long category_id)	Specifies output to a report database
	void	**output**	(string name, Image ptr image)	Specifies output to an image
	void	**queue**	(string script)	Queues a script for parallel execution
	void	**scale_to_dbu=**	(bool en)	Enables or disabled automatic scaling to database unit
[const]	bool	**scale_to_dbu?**		Gets a valid indicating whether automatic scaling to database unit is enabled
[const]	unsigned long	**threads**		Gets the number of threads to use
	void	**threads=**	(unsigned long n)	Specifies the number of threads to use
	void	**tile_border**	(double bx, double by)	Sets the tile border
	void	**tile_origin**	(double xo, double yo)	Sets the tile origin
	void	**tile_size**	(double w, double h)	Sets the tile size
	void	**tiles**	(unsigned long nw, unsigned long nh)	Sets the tile count
	void	**var**	(string name, variant value)	Defines a variable for the tiling processor script

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead

[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const TilingProcessor other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

dbu

Signature: *[const]* double **dbu**

Description: Gets the database unit under which the computations will be done

Python specific notes:
The object exposes a readable attribute 'dbu'. This is the getter.

dbu=

Signature: void **dbu=** (double u)

Description: Sets the database unit under which the computations will be done

All data used within the scripts will be brought to that database unit. If none is given it will be the database unit of the first layout given or 1nm if no layout is specified.

Python specific notes:
The object exposes a writable attribute 'dbu'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new TilingProcessor ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

execute

Signature: void **execute** (string desc)

Description: Runs the job

This method will initiate execution of the queued scripts, once for every tile. The desc is a text shown in the progress bar for example.

frame=

Signature: void **frame=** (const DBox frame)

Description: Sets the layout frame

The layout frame is the box (in micron units) taken into account for computing the tiles if the tile counts are not given. If the layout frame is not set or set to an empty box, the processor will try to derive the frame from the given inputs.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'frame'. This is the setter.

input

(1) Signature: void **input** (string name, const RecursiveShapeIterator iter)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from a recursive shape iterator, hence from a hierarchy of shapes from a layout.

The name specifies the variable under which the input can be used in the scripts.

(2) Signature: void **input** (string name, const RecursiveShapeIterator iter, const ICplxTrans trans)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from a recursive shape iterator, hence from a hierarchy of shapes from a layout. In addition, a transformation can be specified which will be applied to the shapes before they are used.

The name specifies the variable under which the input can be used in the scripts.

(3) Signature: void **input** (string name, const Layout layout, unsigned int cell_index, const LayerInfo lp)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from a layout and the hierarchy below the cell with the given cell index. "lp" is a LayerInfo object specifying the input layer.

The name specifies the variable under which the input can be used in the scripts.

(4) Signature: void **input** (string name, const Layout layout, unsigned int cell_index, unsigned int layer)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from a layout and the hierarchy below the cell with the given cell index. "layer" is the layer index of the input layer.

The name specifies the variable under which the input can be used in the scripts.

(5) Signature: void **input** (string name, const Layout layout, unsigned int cell_index, const LayerInfo lp, const ICplxTrans trans)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from a layout and the hierarchy below the cell with the given cell index. "lp" is a LayerInfo object specifying the input layer. In addition, a transformation can be specified which will be applied to the shapes before they are used.

The name specifies the variable under which the input can be used in the scripts.

(6) Signature: void **input** (string name, const Layout layout, unsigned int cell_index, unsigned int layer, const ICplxTrans trans)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from a layout and the hierarchy below the cell with the given cell index. "layer" is the layer index of the input layer. In addition, a transformation can be specified which will be applied to the shapes before they are used.

The name specifies the variable under which the input can be used in the scripts.

(7) Signature: void **input** (string name, const Region region)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from a Region object. Regions don't always come with a database unit, hence a database unit should be specified with the dbu= method unless a layout object is specified as input too.

Caution: the Region object must stay valid during the lifetime of the tiling processor. Take care to store it in a variable to prevent early destruction of the Region object. Not doing so may crash the application.

The name specifies the variable under which the input can be used in the scripts.

(8) Signature: void **input** (string name, const Region region, const ICplxTrans trans)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from a Region object. Regions don't always come with a database unit, hence a database unit should be specified with the dbu= method unless a layout object is specified as input too.

Caution: the Region object must stay valid during the lifetime of the tiling processor. Take care to store it in a variable to prevent early destruction of the Region object. Not doing so may crash the application.

The name specifies the variable under which the input can be used in the scripts. This variant allows one to specify an additional transformation too. It has been introduced in version 0.23.2.

(9) Signature: void **input** (string name, const Edges edges)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from an Edges object. Edge collections don't always come with a database unit, hence a database unit should be specified with the dbu= method unless a layout object is specified as input too.

Caution: the Edges object must stay valid during the lifetime of the tiling processor. Take care to store it in a variable to prevent early destruction of the Edges object. Not doing so may crash the application.

The name specifies the variable under which the input can be used in the scripts.

(10) Signature: void **input** (string name, const Edges edges, const ICplxTrans trans)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from an Edges object. Edge collections don't always come with a database unit, hence a database unit should be specified with the dbu= method unless a layout object is specified as input too.

Caution: the Edges object must stay valid during the lifetime of the tiling processor. Take care to store it in a variable to prevent early destruction of the Edges object. Not doing so may crash the application.

The name specifies the variable under which the input can be used in the scripts. This variant allows one to specify an additional transformation too. It has been introduced in version 0.23.2.

(11) Signature: void **input** (string name, const EdgePairs edge_pairs)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from an EdgePairs object. Edge pair collections don't always come with a database unit, hence a database unit should be specified with the dbu= method unless a layout object is specified as input too.

Caution: the EdgePairs object must stay valid during the lifetime of the tiling processor. Take care to store it in a variable to prevent early destruction of the EdgePairs object. Not doing so may crash the application.

The name specifies the variable under which the input can be used in the scripts. This variant has been introduced in version 0.27.

(12) Signature: void **input** (string name, const EdgePairs edge_pairs, const ICplxTrans trans)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from an EdgePairs object. Edge pair collections don't always come with a database unit, hence a database unit should be specified with the dbu= method unless a layout object is specified as input too.

Caution: the EdgePairs object must stay valid during the lifetime of the tiling processor. Take care to store it in a variable to prevent early destruction of the EdgePairs object. Not doing so may crash the application.

The name specifies the variable under which the input can be used in the scripts. This variant has been introduced in version 0.27.

(13) Signature: void **input** (string name, const Texts texts)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from an Texts object. Text collections don't always come with a database unit, hence a database unit should be specified with the dbu= method unless a layout object is specified as input too.

Caution: the Texts object must stay valid during the lifetime of the tiling processor. Take care to store it in a variable to prevent early destruction of the Texts object. Not doing so may crash the application.

The name specifies the variable under which the input can be used in the scripts. This variant has been introduced in version 0.27.

(14) Signature: void **input** (string name, const Texts texts, const ICplxTrans trans)

Description: Specifies input for the tiling processor

This method will establish an input channel for the processor. This version receives input from an Texts object. Text collections don't always come with a database unit, hence a database unit should be specified with the dbu= method unless a layout object is specified as input too.

Caution: the Texts object must stay valid during the lifetime of the tiling processor. Take care to store it in a variable to prevent early destruction of the Texts object. Not doing so may crash the application.

The name specifies the variable under which the input can be used in the scripts. This variant has been introduced in version 0.27.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new TilingProcessor ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

output

(1) Signature: void **output** (string name, TileOutputReceiver ptr rec)

Description: Specifies output for the tiling processor

This method will establish an output channel for the processor. For that it registers an output receiver which will receive data from the scripts. The scripts call the _output function to deliver data. "name" will be name of the variable which must be passed to the first argument of the _output function in order to address this channel.

Please note that the tiling processor will destroy the receiver object when it is freed itself. Hence if you need to address the receiver object later, make sure that the processor is still alive, i.e. by assigning the object to a variable.

The following code uses the output receiver. It takes the shapes of a layer from a layout, computes the area of each tile and outputs the area to the custom receiver:

```
layout = ... # the layout
cell = ... # the top cell's index
layout = ... # the input layer

class MyReceiver < RBA::TileOutputReceiver
  def put(ix, iy, tile, obj, dbu, clip)
    puts "got area for tile #{ix+1},#{iy+1}: #{obj.to_s}"
  end
end

tp = RBA::TilingProcessor::new

# register the custom receiver
tp.output("my_receiver", MyReceiver::new)
tp.input("the_input", layout.begin_shapes(cell, layer))
tp.tile_size(100, 100)  # 100x100 um tile size
# The script clips the input at the tile and computes the (merged) area:
tp.queue("_output(my_receiver, (the_input & _tile).area)")
tp.execute("Job description")
```

(2) Signature: void **output** (string name, Layout layout, unsigned int cell, const LayerInfo lp)

Description: Specifies output to a layout layer

name:	The name of the channel
layout:	The layout to which the data is sent
cell:	The index of the cell to which the data is sent
lp:	The layer specification where the output will be sent to

This method will establish an output channel to a layer in a layout. The output sent to that channel will be put into the specified layer and cell. In this version, the layer is specified through a LayerInfo object, i.e. layer and datatype number. If no such layer exists, it will be created.

The name is the name which must be used in the _output function of the scripts in order to address that channel.

(3) Signature: void **output** (string name, Layout layout, unsigned int cell, unsigned int layer_index)

Description: Specifies output to a layout layer

name:	The name of the channel
layout:	The layout to which the data is sent
cell:	The index of the cell to which the data is sent

layer_index: The layer index where the output will be sent to

This method will establish an output channel to a layer in a layout. The output sent to that channel will be put into the specified layer and cell. In this version, the layer is specified through a layer index, hence it must be created before.

The name is the name which must be used in the _output function of the scripts in order to address that channel.

(4) Signature: void **output** (string name, Region region)

Description: Specifies output to a Region object

name: The name of the channel

region: The Region object to which the data is sent

This method will establish an output channel to a Region object. The output sent to that channel will be put into the specified region.

The name is the name which must be used in the _output function of the scripts in order to address that channel. Edges sent to this channel are discarded. Edge pairs are converted to polygons.

(5) Signature: void **output** (string name, Edges edges)

Description: Specifies output to an Edges object

name: The name of the channel

edges: The Edges object to which the data is sent

This method will establish an output channel to an Edges object. The output sent to that channel will be put into the specified edge collection. 'Solid' objects such as polygons will be converted to edges by resolving their hulls into edges. Edge pairs are resolved into single edges.

The name is the name which must be used in the _output function of the scripts in order to address that channel.

(6) Signature: void **output** (string name, EdgePairs edge_pairs)

Description: Specifies output to an EdgePairs object

name: The name of the channel

edge_pairs: The EdgePairs object to which the data is sent

This method will establish an output channel to an EdgePairs object. The output sent to that channel will be put into the specified edge pair collection. Only EdgePair objects are accepted. Other objects are discarded.

The name is the name which must be used in the _output function of the scripts in order to address that channel.

(7) Signature: void **output** (string name, Texts texts)

Description: Specifies output to an Texts object

name: The name of the channel

texts: The Texts object to which the data is sent

This method will establish an output channel to an Texts object. The output sent to that channel will be put into the specified edge pair collection. Only Text objects are accepted. Other objects are discarded.

The name is the name which must be used in the _output function of the scripts in order to address that channel.

This variant has been introduced in version 0.27.

(8) Signature: void **output** (string name, double ptr sum)

Description: Specifies output to single value

This method will establish an output channel which sums up float data delivered by calling the _output function. In order to specify the target for the data, a Value object must be provided for the "sum" parameter.

The name is the name which must be used in the _output function of the scripts in order to address that channel.

(9) Signature: void **output** (string name, ReportDatabase rdb, unsigned long cell_id, unsigned long category_id)

Description: Specifies output to a report database

This method will establish an output channel for the processor. The output sent to that channel will be put into the report database given by the "rdb" parameter. "cell_id" specifies the cell and "category_id" the category to use.

The name is the name which must be used in the _output function of the scripts in order to address that channel.

(10) Signature: void **output** (string name, Image ptr image)

Description: Specifies output to an image

This method will establish an output channel which delivers float data to image data. The image is a monochrome image where each pixel corresponds to a single tile. This method for example is useful to collect density information into an image. The image is configured such that each pixel covers one tile.

The name is the name which must be used in the _output function of the scripts in order to address that channel.

queue

Signature: void **queue** (string script)

Description: Queues a script for parallel execution

With this method, scripts are registered that are executed in parallel on each tile. The scripts have "Expressions" syntax and can make use of several predefined variables and functions. See the TilingProcessor class description for details.

scale_to_dbu=

Signature: void **scale_to_dbu=** (bool en)

Description: Enables or disabled automatic scaling to database unit

If automatic scaling to database unit is enabled, the input is automatically scaled to the database unit set inside the tile processor. This is the default.

This method has been introduced in version 0.23.2.

Python specific notes:
The object exposes a writable attribute 'scale_to_dbu'. This is the setter.

scale_to_dbu?

Signature: *[const]* bool **scale_to_dbu?**

Description: Gets a valid indicating whether automatic scaling to database unit is enabled

This method has been introduced in version 0.23.2.

Python specific notes:
The object exposes a readable attribute 'scale_to_dbu'. This is the getter.

threads

Signature: *[const]* unsigned long **threads**

Description: Gets the number of threads to use

Python specific notes:

The object exposes a readable attribute 'threads'. This is the getter.

threads=

Signature: void **threads=** (unsigned long n)

Description: Specifies the number of threads to use

Python specific notes:
The object exposes a writable attribute 'threads'. This is the setter.

tile_border

Signature: void **tile_border** (double bx, double by)

Description: Sets the tile border

Specifies the tile border. The border is a margin that is considered when fetching shapes. By specifying a border you can fetch shapes into the tile's data which are outside the tile but still must be considered in the computations (i.e. because they might grow into the tile).

The tile border is given in micron.

tile_origin

Signature: void **tile_origin** (double xo, double yo)

Description: Sets the tile origin

Specifies the origin (lower left corner) of the tile field. If no origin is specified, the tiles are centered to the layout's bounding box. Giving the origin together with the tile count and dimensions gives full control over the tile array.

The tile origin is given in micron.

tile_size

Signature: void **tile_size** (double w, double h)

Description: Sets the tile size

Specifies the size of the tiles to be used. If no tile size is specified, tiling won't be used and all computations will be done on the whole layout.

The tile size is given in micron.

tiles

Signature: void **tiles** (unsigned long nw, unsigned long nh)

Description: Sets the tile count

Specifies the number of tiles to be used. If no tile number is specified, the number of tiles required is computed from the layout's dimensions and the tile size. If a number is given, but no tile size, the tile size will be computed from the layout's dimensions.

var

Signature: void **var** (string name, variant value)

Description: Defines a variable for the tiling processor script

The name specifies the variable under which the value can be used in the scripts.

4.105. API reference - Class Trans

Notation used in Ruby API documentation

Module: db

Description: A simple transformation

Simple transformations only provide rotations about angles which a multiples of 90 degree. Together with the mirror options, this results in 8 distinct orientations (fixpoint transformations). These can be combined with a displacement which is applied after the rotation/mirror. This version acts on integer coordinates. A version for floating-point coordinates is DTrans.

Here are some examples for using the Trans class:

```
t = RBA::Trans::new(0, 100)  # displacement by 100 DBU in y direction
# the inverse: -> "r0 0,-100"
t.inverted.to_s
# concatenation: -> "r90 -100,0"
(RBA::Trans::R90 * t).to_s
# apply to a point: -> "0,100"
RBA::Trans::R90.trans(RBA::Point::new(100, 0))
```

See The Database API for more details about the database objects.

Public constructors

new Trans ptr	**new**	(const DTrans dtrans)	Creates an integer coordinate transformation from a floating-point coordinate transformation
new Trans ptr	**new**		Creates a unit transformation
new Trans ptr	**new**	(const Trans c, const Vector u = 0,0)	Creates a transformation from another transformation plus a displacement
new Trans ptr	**new**	(const Trans c, int x = 0, int y = 0)	Creates a transformation from another transformation plus a displacement
new Trans ptr	**new**	(int rot = 0, bool mirrx = false, const Vector u = 0,0)	Creates a transformation using angle and mirror flag
new Trans ptr	**new**	(int rot = 0, bool mirrx = false, int x = 0, int y = 0)	Creates a transformation using angle and mirror flag and two coordinate values for displacement
new Trans ptr	**new**	(const Vector u)	Creates a transformation using a displacement only
new Trans ptr	**new**	(int x, int y)	Creates a transformation using a displacement given as two coordinates

Public methods

[const]	bool	**!=**	(const Trans other)	Tests for inequality
[const]	unsigned int	*****	(unsigned int d)	Transforms a distance

[const]	Point	*****	(const Point p)	Transforms a point
[const]	Vector	*****	(const Vector v)	Transforms a vector
[const]	Box	*****	(const Box box)	Transforms a box
[const]	Edge	*****	(const Edge edge)	Transforms an edge
[const]	Polygon	*****	(const Polygon polygon)	Transforms a polygon
[const]	Path	*****	(const Path path)	Transforms a path
[const]	Text	*****	(const Text text)	Transforms a text
[const]	Trans	*****	(const Trans t)	Returns the concatenated transformation
[const]	bool	**<**	(const Trans other)	Provides a 'less' criterion for sorting
[const]	bool	**==**	(const Trans other)	Tests for equality
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	int	**angle**		Gets the angle in units of 90 degree
	void	**angle=**	(int a)	Sets the angle in units of 90 degree
	void	**assign**	(const Trans other)	Assigns another object to self
[const]	unsigned int	**ctrans**	(unsigned int d)	Transforms a distance
[const]	Vector	**disp**		Gets to the displacement vector
	void	**disp=**	(const Vector u)	Sets the displacement
[const]	new Trans ptr	**dup**		Creates a copy of self
[const]	unsigned long	**hash**		Computes a hash value
	Trans	**invert**		Inverts the transformation (in place)
[const]	Trans	**inverted**		Returns the inverted transformation
[const]	bool	**is_mirror?**		Gets the mirror flag

	void	**mirror=**	(bool m)	Sets the mirror flag
[const]	int	**rot**		Gets the angle/mirror code
	void	**rot=**	(int r)	Sets the angle/mirror code
[const]	DTrans	**to_dtype**	(double dbu = 1)	Converts the transformation to a floating-point coordinate transformation
[const]	string	**to_s**	(double dbu = 0)	String conversion
[const]	Point	**trans**	(const Point p)	Transforms a point
[const]	Vector	**trans**	(const Vector v)	Transforms a vector
[const]	Box	**trans**	(const Box box)	Transforms a box
[const]	Edge	**trans**	(const Edge edge)	Transforms an edge
[const]	Polygon	**trans**	(const Polygon polygon)	Transforms a polygon
[const]	Path	**trans**	(const Path path)	Transforms a path
[const]	Text	**trans**	(const Text text)	Transforms a text

Public static methods and constants

Trans	**M0**		A constant giving "mirrored at the x-axis" transformation
Trans	**M135**		A constant giving "mirrored at the 135 degree axis" transformation
Trans	**M45**		A constant giving "mirrored at the 45 degree axis" transformation
Trans	**M90**		A constant giving "mirrored at the y (90 degree) axis" transformation
Trans	**R0**		A constant giving "unrotated" (unit) transformation
Trans	**R180**		A constant giving "rotated by 180 degree counterclockwise" transformation
Trans	**R270**		A constant giving "rotated by 270 degree counterclockwise" transformation
Trans	**R90**		A constant giving "rotated by 90 degree counterclockwise" transformation
new Trans ptr	**from_s**	(string s)	Creates a transformation from a string

Deprecated methods (protected, public, static, non-static and constructors)

void	**create**		Use of this method is deprecated. Use _create instead

	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new Trans ptr	**from_dtrans**	(const DTrans dtrans)	Use of this method is deprecated. Use new instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const Trans other)

Description: Tests for inequality

*****

(1) Signature: *[const]* unsigned int ***** (unsigned int d)

Description: Transforms a distance

d: The distance to transform

Returns: The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* Point ***** (const Point p)

Description: Transforms a point

p: The point to transform

Returns: The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* Vector ***** (const Vector v)

Description: Transforms a vector

v: The vector to transform

Returns: The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* Box ***** (const Box box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* Edge ***** (const Edge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* Polygon ***** (const Polygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform
Returns:	The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* Path ***** (const Path path)

Description: Transforms a path

path:	The path to transform
Returns:	The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(8) Signature: *[const]* Text ***** (const Text text)

Description: Transforms a text

text:	The text to transform
Returns:	The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(9) Signature: *[const]* Trans ***** (const Trans t)

Description: Returns the concatenated transformation

t:	The transformation to apply before
Returns:	The modified transformation

The * operator returns self*t ("t is applied before this transformation").

<

Signature: *[const]* bool **<** (const Trans other)

Description: Provides a 'less' criterion for sorting

This method is provided to implement a sorting order. The definition of 'less' is opaque and might change in future versions.

==

Signature: *[const]* bool **==** (const Trans other)

Description: Tests for equality

M0

Signature: *[static]* Trans **M0**

Description: A constant giving "mirrored at the x-axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M0'. This is the getter.

M135

Signature: *[static]* Trans **M135**

Description: A constant giving "mirrored at the 135 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M135'. This is the getter.

M45

Signature: *[static]* Trans **M45**

Description: A constant giving "mirrored at the 45 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M45'. This is the getter.

M90

Signature: *[static]* Trans **M90**

Description: A constant giving "mirrored at the y (90 degree) axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M90'. This is the getter.

R0

Signature: *[static]* Trans **R0**

Description: A constant giving "unrotated" (unit) transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R0'. This is the getter.

R180

Signature: *[static]* Trans **R180**

Description: A constant giving "rotated by 180 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R180'. This is the getter.

R270

Signature: *[static]* Trans **R270**

Description: A constant giving "rotated by 270 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R270'. This is the getter.

R90

Signature: *[static]* Trans **R90**

Description: A constant giving "rotated by 90 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R90'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

angle

Signature: *[const]* int **angle**

Description: Gets the angle in units of 90 degree

This value delivers the rotation component. In addition, a mirroring at the x axis may be applied before if the is_mirror? property is true.

Python specific notes:
The object exposes a readable attribute 'angle'. This is the getter.

angle=

Signature: void **angle=** (int a)

Description: Sets the angle in units of 90 degree

a:	The new angle

This method was introduced in version 0.20.

Python specific notes:
The object exposes a writable attribute 'angle'. This is the setter.

assign

Signature: void **assign** (const Trans other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

ctrans

Signature: *[const]* unsigned int **ctrans** (unsigned int d)

Description: Transforms a distance

d:	The distance to transform
Returns:	The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disp

Signature: *[const]* Vector **disp**

Description: Gets to the displacement vector

Staring with version 0.25 the displacement type is a vector.

Python specific notes:
The object exposes a readable attribute 'disp'. This is the getter.

disp=

Signature: void **disp=** (const Vector u)

Description: Sets the displacement

u: The new displacement

This method was introduced in version 0.20. Staring with version 0.25 the displacement type is a vector.

Python specific notes:
The object exposes a writable attribute 'disp'. This is the setter.

dup

Signature: *[const]* new Trans ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_dtrans

Signature: *[static]* new Trans ptr **from_dtrans** (const DTrans dtrans)

Description: Creates an integer coordinate transformation from a floating-point coordinate transformation

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dtrans'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new Trans ptr **from_s** (string s)

Description: Creates a transformation from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given transformation. This method enables transformations as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

invert

Signature: Trans **invert**

Description: Inverts the transformation (in place)

Returns: The inverted transformation

Inverts the transformation and replaces this object by the inverted one.

inverted

Signature: *[const]* Trans **inverted**

Description: Returns the inverted transformation

Returns: The inverted transformation

Returns the inverted transformation

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_mirror?

Signature: *[const]* bool **is_mirror?**

Description: Gets the mirror flag

If this property is true, the transformation is composed of a mirroring at the x-axis followed by a rotation by the angle given by the angle property.

Python specific notes:
The object exposes a readable attribute 'mirror'. This is the getter.

mirror=

Signature: void **mirror=** (bool m)

Description: Sets the mirror flag

m: The new mirror flag

"mirroring" describes a reflection at the x-axis which is included in the transformation prior to rotation.

This method was introduced in version 0.20.

Python specific notes:
The object exposes a writable attribute 'mirror'. This is the setter.

new

(1) Signature: *[static]* new Trans ptr **new** (const DTrans dtrans)

Description: Creates an integer coordinate transformation from a floating-point coordinate transformation

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_dtrans'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Trans ptr **new**

Description: Creates a unit transformation

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Trans ptr **new** (const Trans c, const Vector u = 0,0)

Description: Creates a transformation from another transformation plus a displacement

c:	The original transformation
u:	The Additional displacement

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Trans ptr **new** (const Trans c, int x = 0, int y = 0)

Description: Creates a transformation from another transformation plus a displacement

c:	The original transformation
x:	The Additional displacement (x)
y:	The Additional displacement (y)

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new Trans ptr **new** (int rot = 0, bool mirrx = false, const Vector u = 0,0)

Description: Creates a transformation using angle and mirror flag

rot:	The rotation in units of 90 degree
mirrx:	True, if mirrored at x axis
u:	The displacement

The sequence of operations is: mirroring at x axis, rotation, application of displacement.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new Trans ptr **new** (int rot = 0, bool mirrx = false, int x = 0, int y = 0)

Description: Creates a transformation using angle and mirror flag and two coordinate values for displacement

rot:	The rotation in units of 90 degree
mirrx:	True, if mirrored at x axis
x:	The horizontal displacement
y:	The vertical displacement

The sequence of operations is: mirroring at x axis, rotation, application of displacement.

Python specific notes:

This method is the default initializer of the object.

(7) Signature: *[static]* new Trans ptr **new** (const Vector u)

Description: Creates a transformation using a displacement only

u: The displacement

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new Trans ptr **new** (int x, int y)

Description: Creates a transformation using a displacement given as two coordinates

x: The horizontal displacement

y: The vertical displacement

Python specific notes:
This method is the default initializer of the object.

rot

Signature: *[const]* int **rot**

Description: Gets the angle/mirror code

The angle/mirror code is one of the constants R0, R90, R180, R270, M0, M45, M90 and M135. rx is the rotation by an angle of x counter clockwise. mx is the mirroring at the axis given by the angle x (to the x-axis).

Python specific notes:
The object exposes a readable attribute 'rot'. This is the getter.

rot=

Signature: void **rot=** (int r)

Description: Sets the angle/mirror code

r: The new angle/rotation code (see rot property)

This method was introduced in version 0.20.

Python specific notes:
The object exposes a writable attribute 'rot'. This is the setter.

to_dtype

Signature: *[const]* DTrans **to_dtype** (double dbu = 1)

Description: Converts the transformation to a floating-point coordinate transformation

The database unit can be specified to translate the integer-coordinate transformation into a floating-point coordinate transformation in micron units. The database unit is basically a scaling factor.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: String conversion

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

trans

(1) Signature: *[const]* Point **trans** (const Point p)

Description: Transforms a point

p:	The point to transform
Returns:	The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* Vector **trans** (const Vector v)

Description: Transforms a vector

v:	The vector to transform
Returns:	The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* Box **trans** (const Box box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* Edge **trans** (const Edge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* Polygon **trans** (const Polygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform
Returns:	The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* Path **trans** (const Path path)

Description: Transforms a path

path:	The path to transform
Returns:	The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* Text **trans** (const Text text)

Description: Transforms a text

text:	The text to transform
Returns:	The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

4.106. API reference - Class DTrans

Notation used in Ruby API documentation

Module: db

Description: A simple transformation

Simple transformations only provide rotations about angles which a multiples of 90 degree. Together with the mirror options, this results in 8 distinct orientations (fixpoint transformations). These can be combined with a displacement which is applied after the rotation/mirror. This version acts on floating-point coordinates. A version for integer coordinates is Trans.

Here are some examples for using the DTrans class:

```
t = RBA::DTrans::new(0, 100)  # displacement by 100 DBU in y direction
# the inverse: -> "r0 0,-100"
t.inverted.to_s
# concatenation: -> "r90 -100,0"
(RBA::DTrans::new(RBA::DTrans::R90) * t).to_s
# apply to a point: -> "0,100"
RBA::DTrans::new(RBA::DTrans::R90).trans(RBA::DPoint::new(100, 0))
```

See The Database API for more details about the database objects.

Public constructors

new DTrans ptr	**new**	(const Trans trans)	Creates a floating-point coordinate transformation from an integer coordinate transformation
new DTrans ptr	**new**		Creates a unit transformation
new DTrans ptr	**new**	(const DTrans c, const DVector u = 0,0)	Creates a transformation from another transformation plus a displacement
new DTrans ptr	**new**	(const DTrans c, double x = 0, double y = 0)	Creates a transformation from another transformation plus a displacement
new DTrans ptr	**new**	(int rot = 0, bool mirrx = false, const DVector u = 0,0)	Creates a transformation using angle and mirror flag
new DTrans ptr	**new**	(int rot = 0, bool mirrx = false, double x = 0, double y = 0)	Creates a transformation using angle and mirror flag and two coordinate values for displacement
new DTrans ptr	**new**	(const DVector u)	Creates a transformation using a displacement only
new DTrans ptr	**new**	(double x, double y)	Creates a transformation using a displacement given as two coordinates

Public methods

[const]	bool	**!=**	(const DTrans other)	Tests for inequality

[const]	double	*	(double d)	Transforms a distance
[const]	DPoint	*	(const DPoint p)	Transforms a point
[const]	DVector	*	(const DVector v)	Transforms a vector
[const]	DBox	*	(const DBox box)	Transforms a box
[const]	DEdge	*	(const DEdge edge)	Transforms an edge
[const]	DPolygon	*	(const DPolygon polygon)	Transforms a polygon
[const]	DPath	*	(const DPath path)	Transforms a path
[const]	DText	*	(const DText text)	Transforms a text
[const]	DTrans	*	(const DTrans t)	Returns the concatenated transformation
[const]	bool	<	(const DTrans other)	Provides a 'less' criterion for sorting
[const]	bool	==	(const DTrans other)	Tests for equality
	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	int	angle		Gets the angle in units of 90 degree
	void	angle=	(int a)	Sets the angle in units of 90 degree
	void	assign	(const DTrans other)	Assigns another object to self
[const]	double	ctrans	(double d)	Transforms a distance
[const]	DVector	disp		Gets to the displacement vector
	void	disp=	(const DVector u)	Sets the displacement
[const]	new DTrans ptr	dup		Creates a copy of self
[const]	unsigned long	hash		Computes a hash value
	DTrans	invert		Inverts the transformation (in place)
[const]	DTrans	inverted		Returns the inverted transformation

[const]	bool	**is_mirror?**		Gets the mirror flag
	void	**mirror=**	(bool m)	Sets the mirror flag
[const]	int	**rot**		Gets the angle/mirror code
	void	**rot=**	(int r)	Sets the angle/mirror code
[const]	Trans	**to_itype**	(double dbu = 1)	Converts the transformation to an integer coordinate transformation
[const]	string	**to_s**	(double dbu = 0)	String conversion
[const]	DPoint	**trans**	(const DPoint p)	Transforms a point
[const]	DVector	**trans**	(const DVector v)	Transforms a vector
[const]	DBox	**trans**	(const DBox box)	Transforms a box
[const]	DEdge	**trans**	(const DEdge edge)	Transforms an edge
[const]	DPolygon	**trans**	(const DPolygon polygon)	Transforms a polygon
[const]	DPath	**trans**	(const DPath path)	Transforms a path
[const]	DText	**trans**	(const DText text)	Transforms a text

Public static methods and constants

DTrans	**M0**		A constant giving "mirrored at the x-axis" transformation
DTrans	**M135**		A constant giving "mirrored at the 135 degree axis" transformation
DTrans	**M45**		A constant giving "mirrored at the 45 degree axis" transformation
DTrans	**M90**		A constant giving "mirrored at the y (90 degree) axis" transformation
DTrans	**R0**		A constant giving "unrotated" (unit) transformation
DTrans	**R180**		A constant giving "rotated by 180 degree counterclockwise" transformation
DTrans	**R270**		A constant giving "rotated by 270 degree counterclockwise" transformation
DTrans	**R90**		A constant giving "rotated by 90 degree counterclockwise" transformation
new DTrans ptr	**from_s**	(string s)	Creates a transformation from a string

Deprecated methods (protected, public, static, non-static and constructors)

void	**create**		Use of this method is deprecated. Use _create instead

	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new DTrans ptr	**from_itrans**	(const Trans trans)	Use of this method is deprecated. Use new instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const DTrans other)

Description: Tests for inequality

*****

(1) Signature: *[const]* double ***** (double d)

Description: Transforms a distance

d: The distance to transform

Returns: The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* DPoint ***** (const DPoint p)

Description: Transforms a point

p: The point to transform

Returns: The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* DVector ***** (const DVector v)

Description: Transforms a vector

v: The vector to transform

Returns: The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* DBox ***** (const DBox box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* DEdge ***** (const DEdge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* DPolygon ***** (const DPolygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform
Returns:	The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* DPath ***** (const DPath path)

Description: Transforms a path

path:	The path to transform
Returns:	The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(8) Signature: *[const]* DText ***** (const DText text)

Description: Transforms a text

text:	The text to transform
Returns:	The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(9) Signature: *[const]* DTrans ***** (const DTrans t)

Description: Returns the concatenated transformation

t:	The transformation to apply before
Returns:	The modified transformation

The * operator returns self*t ("t is applied before this transformation").

<

Signature: *[const]* bool **<** (const DTrans other)

Description: Provides a 'less' criterion for sorting

This method is provided to implement a sorting order. The definition of 'less' is opaque and might change in future versions.

==

Signature: *[const]* bool **==** (const DTrans other)

Description: Tests for equality

M0

Signature: *[static]* DTrans **M0**

Description: A constant giving "mirrored at the x-axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M0'. This is the getter.

M135

Signature: *[static]* DTrans **M135**

Description: A constant giving "mirrored at the 135 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M135'. This is the getter.

M45

Signature: *[static]* DTrans **M45**

Description: A constant giving "mirrored at the 45 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M45'. This is the getter.

M90

Signature: *[static]* DTrans **M90**

Description: A constant giving "mirrored at the y (90 degree) axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M90'. This is the getter.

R0

Signature: *[static]* DTrans **R0**

Description: A constant giving "unrotated" (unit) transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R0'. This is the getter.

R180

Signature: *[static]* DTrans **R180**

Description: A constant giving "rotated by 180 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R180'. This is the getter.

R270

Signature: *[static]* DTrans **R270**

Description: A constant giving "rotated by 270 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R270'. This is the getter.

R90

Signature: *[static]* DTrans **R90**

Description: A constant giving "rotated by 90 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R90'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

angle

Signature: *[const]* int **angle**

Description: Gets the angle in units of 90 degree

This value delivers the rotation component. In addition, a mirroring at the x axis may be applied before if the is_mirror? property is true.

Python specific notes:
The object exposes a readable attribute 'angle'. This is the getter.

angle=

Signature: void **angle=** (int a)

Description: Sets the angle in units of 90 degree

a:	The new angle

This method was introduced in version 0.20.

Python specific notes:
The object exposes a writable attribute 'angle'. This is the setter.

assign

Signature: void **assign** (const DTrans other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

ctrans

Signature: *[const]* double **ctrans** (double d)

Description: Transforms a distance

d:	The distance to transform
Returns:	The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disp

Signature: *[const]* DVector **disp**

Description: Gets to the displacement vector

Staring with version 0.25 the displacement type is a vector.

Python specific notes:
The object exposes a readable attribute 'disp'. This is the getter.

disp=

Signature: void **disp=** (const DVector u)

Description: Sets the displacement

u: The new displacement

This method was introduced in version 0.20. Staring with version 0.25 the displacement type is a vector.

Python specific notes:
The object exposes a writable attribute 'disp'. This is the setter.

dup

Signature: *[const]* new DTrans ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_itrans

Signature: *[static]* new DTrans ptr **from_itrans** (const Trans trans)

Description: Creates a floating-point coordinate transformation from an integer coordinate transformation

Use of this method is deprecated. Use new instead

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_itrans'.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new DTrans ptr **from_s** (string s)

Description: Creates a transformation from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given transformation. This method enables transformations as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

invert

Signature: DTrans **invert**

Description: Inverts the transformation (in place)

Returns: The inverted transformation

Inverts the transformation and replaces this object by the inverted one.

inverted

Signature: *[const]* DTrans **inverted**

Description: Returns the inverted transformation

Returns: The inverted transformation

Returns the inverted transformation

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_mirror?

Signature: *[const]* bool **is_mirror?**

Description: Gets the mirror flag

If this property is true, the transformation is composed of a mirroring at the x-axis followed by a rotation by the angle given by the angle property.

Python specific notes:
The object exposes a readable attribute 'mirror'. This is the getter.

mirror=

Signature: void **mirror=** (bool m)

Description: Sets the mirror flag

m: The new mirror flag

"mirroring" describes a reflection at the x-axis which is included in the transformation prior to rotation.

This method was introduced in version 0.20.

Python specific notes:
The object exposes a writable attribute 'mirror'. This is the setter.

new

(1) Signature: *[static]* new DTrans ptr **new** (const Trans trans)

Description: Creates a floating-point coordinate transformation from an integer coordinate transformation

This constructor has been introduced in version 0.25 and replaces the previous static method 'from_itrans'.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DTrans ptr **new**

Description: Creates a unit transformation

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new DTrans ptr **new** (const DTrans c, const DVector u = 0,0)

Description: Creates a transformation from another transformation plus a displacement

c:	The original transformation
u:	The Additional displacement

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new DTrans ptr **new** (const DTrans c, double x = 0, double y = 0)

Description: Creates a transformation from another transformation plus a displacement

c:	The original transformation
x:	The Additional displacement (x)
y:	The Additional displacement (y)

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new DTrans ptr **new** (int rot = 0, bool mirrx = false, const DVector u = 0,0)

Description: Creates a transformation using angle and mirror flag

rot:	The rotation in units of 90 degree
mirrx:	True, if mirrored at x axis
u:	The displacement

The sequence of operations is: mirroring at x axis, rotation, application of displacement.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new DTrans ptr **new** (int rot = 0, bool mirrx = false, double x = 0, double y = 0)

Description: Creates a transformation using angle and mirror flag and two coordinate values for displacement

rot:	The rotation in units of 90 degree
mirrx:	True, if mirrored at x axis
x:	The horizontal displacement
y:	The vertical displacement

The sequence of operations is: mirroring at x axis, rotation, application of displacement.

Python specific notes:

This method is the default initializer of the object.

(7) Signature: *[static]* new DTrans ptr **new** (const DVector u)

Description: Creates a transformation using a displacement only

u: The displacement

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new DTrans ptr **new** (double x, double y)

Description: Creates a transformation using a displacement given as two coordinates

x: The horizontal displacement

y: The vertical displacement

Python specific notes:
This method is the default initializer of the object.

rot

Signature: *[const]* int **rot**

Description: Gets the angle/mirror code

The angle/mirror code is one of the constants R0, R90, R180, R270, M0, M45, M90 and M135. rx is the rotation by an angle of x counter clockwise. mx is the mirroring at the axis given by the angle x (to the x-axis).

Python specific notes:
The object exposes a readable attribute 'rot'. This is the getter.

rot=

Signature: void **rot=** (int r)

Description: Sets the angle/mirror code

r: The new angle/rotation code (see rot property)

This method was introduced in version 0.20.

Python specific notes:
The object exposes a writable attribute 'rot'. This is the setter.

to_itype

Signature: *[const]* Trans **to_itype** (double dbu = 1)

Description: Converts the transformation to an integer coordinate transformation

The database unit can be specified to translate the floating-point coordinate transformation in micron units to an integer-coordinate transformation in database units. The transformation's' coordinates will be divided by the database unit.

This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: String conversion

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

trans

(1) Signature: *[const]* DPoint **trans** (const DPoint p)

Description: Transforms a point

p:	The point to transform
Returns:	The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* DVector **trans** (const DVector v)

Description: Transforms a vector

v:	The vector to transform
Returns:	The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* DBox **trans** (const DBox box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* DEdge **trans** (const DEdge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* DPolygon **trans** (const DPolygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform
Returns:	The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:

This method also implements '__rmul__'.

(6) Signature: *[const]* DPath **trans** (const DPath path)

Description: Transforms a path

path:	The path to transform
Returns:	The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* DText **trans** (const DText text)

Description: Transforms a text

text:	The text to transform
Returns:	The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

4.107. API reference - Class DCplxTrans

Notation used in Ruby API documentation

Module: db

Description: A complex transformation

A complex transformation provides magnification, mirroring at the x-axis, rotation by an arbitrary angle and a displacement. This is also the order, the operations are applied.

A complex transformation provides a superset of the simple transformation. In many applications, a complex transformation computes floating-point coordinates to minimize rounding effects. This version can transform floating-point coordinate objects.

Complex transformations are extensions of the simple transformation classes (DTrans in that case) and behave similar.

Transformations can be used to transform points or other objects. Transformations can be combined with the '*' operator to form the transformation which is equivalent to applying the second and then the first. Here is some code:

```
# Create a transformation that applies a magnification of 1.5, a rotation by 90 degree
# and displacement of 10 in x and 20 units in y direction:
t = RBA::CplxTrans::new(1.5, 90, false, 10.0, 20.0)
t.to_s            # r90 *1.5 10,20
# compute the inverse:
t.inverted.to_s   # r270 *0.666666667 -13,7
# Combine with another displacement (applied after that):
(RBA::CplxTrans::new(5, 5) * t).to_s    # r90 *1.5 15,25
# Transform a point:
t.trans(RBA::Point::new(100, 200)).to_s # -290,170
```

See The Database API for more details about the database objects.

Public constructors

new DCplxTrans ptr	**new**	(const CplxTrans trans, double dbu = 1)	Creates a floating-point coordinate transformation from another coordinate flavour
new DCplxTrans ptr	**new**	(const ICplxTrans trans, double dbu = 1)	Creates a floating-point coordinate transformation from another coordinate flavour
new DCplxTrans ptr	**new**	(const VCplxTrans trans, double dbu = 1)	Creates a floating-point coordinate transformation from another coordinate flavour
new DCplxTrans ptr	**new**		Creates a unit transformation
new DCplxTrans ptr	**new**	(const DCplxTrans c, double mag = 1, const DVector u = 0,0)	Creates a transformation from another transformation plus a magnification and displacement
new DCplxTrans ptr	**new**	(const DCplxTrans c, double mag = 1, double x = 0, double y = 0)	Creates a transformation from another transformation plus a magnification and displacement
new DCplxTrans ptr	**new**	(double x, double y)	Creates a transformation from a x and y displacement
new DCplxTrans ptr	**new**	(const DTrans t, double mag = 1)	Creates a transformation from a simple transformation and a magnification

new DCplxTrans ptr	new	(const DVector u)	Creates a transformation from a displacement
new DCplxTrans ptr	new	(double mag = 1, double rot = 0, bool mirrx = false, const DVector u = 0,0)	Creates a transformation using magnification, angle, mirror flag and displacement
new DCplxTrans ptr	new	(double mag = 1, double rot = 0, bool mirrx = false, double x = 0, double y = 0)	Creates a transformation using magnification, angle, mirror flag and displacement

Public methods

[const]	bool	!=	(const DCplxTrans other)	Tests for inequality
[const]	CplxTrans	*	(const CplxTrans t)	Multiplication (concatenation) of transformations
[const]	double	*	(double d)	Transforms a distance
[const]	DPoint	*	(const DPoint p)	Transforms a point
[const]	DVector	*	(const DVector p)	Transforms a vector
[const]	DBox	*	(const DBox box)	Transforms a box
[const]	DEdge	*	(const DEdge edge)	Transforms an edge
[const]	DPolygon	*	(const DPolygon polygon)	Transforms a polygon
[const]	DPath	*	(const DPath path)	Transforms a path
[const]	DText	*	(const DText text)	Transforms a text
[const]	DCplxTrans	*	(const DCplxTrans t)	Returns the concatenated transformation
[const]	bool	<	(const DCplxTrans other)	Provides a 'less' criterion for sorting
[const]	bool	==	(const DCplxTrans other)	Tests for equality
	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference

	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	double	angle		Gets the angle
	void	angle=	(double a)	Sets the angle
	void	assign	(const DCplxTrans other)	Assigns another object to self
[const]	double	ctrans	(double d)	Transforms a distance
[const]	DVector	disp		Gets the displacement
	void	disp=	(const DVector u)	Sets the displacement
[const]	new DCplxTrans ptr	dup		Creates a copy of self
[const]	unsigned long	hash		Computes a hash value
	DCplxTrans	invert		Inverts the transformation (in place)
[const]	DCplxTrans	inverted		Returns the inverted transformation
[const]	bool	is_complex?		Returns true if the transformation is a complex one
[const]	bool	is_mag?		Tests, if the transformation is a magnifying one
[const]	bool	is_mirror?		Gets the mirror flag
[const]	bool	is_ortho?		Tests, if the transformation is an orthogonal transformation
[const]	bool	is_unity?		Tests, whether this is a unit transformation
[const]	double	mag		Gets the magnification
	void	mag=	(double m)	Sets the magnification
	void	mirror=	(bool m)	Sets the mirror flag
[const]	int	rot		Returns the respective simple transformation equivalent rotation code if possible
[const]	DTrans	s_trans		Extracts the simple transformation part
[const]	string	to_s	(bool lazy = false, double dbu = 0)	String conversion
[const]	DPoint	trans	(const DPoint p)	Transforms a point

[const]	DVector	**trans**	(const DVector p)	Transforms a vector
[const]	DBox	**trans**	(const DBox box)	Transforms a box
[const]	DEdge	**trans**	(const DEdge edge)	Transforms an edge
[const]	DPolygon	**trans**	(const DPolygon polygon)	Transforms a polygon
[const]	DPath	**trans**	(const DPath path)	Transforms a path
[const]	DText	**trans**	(const DText text)	Transforms a text

Public static methods and constants

	DCplxTrans	**M0**		A constant giving "mirrored at the x-axis" transformation
	DCplxTrans	**M135**		A constant giving "mirrored at the 135 degree axis" transformation
	DCplxTrans	**M45**		A constant giving "mirrored at the 45 degree axis" transformation
	DCplxTrans	**M90**		A constant giving "mirrored at the y (90 degree) axis" transformation
	DCplxTrans	**R0**		A constant giving "unrotated" (unit) transformation
	DCplxTrans	**R180**		A constant giving "rotated by 180 degree counterclockwise" transformation
	DCplxTrans	**R270**		A constant giving "rotated by 270 degree counterclockwise" transformation
	DCplxTrans	**R90**		A constant giving "rotated by 90 degree counterclockwise" transformation
	new DCplxTrans ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new DCplxTrans ptr	**from_itrans**	(const CplxTrans trans, double dbu = 1)	Use of this method is deprecated. Use new instead

[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	ICplxTrans	**to_itrans**	(double dbu = 1)	Use of this method is deprecated
[const]	CplxTrans	**to_trans**	(double dbu = 1)	Use of this method is deprecated
[const]	VCplxTrans	**to_vtrans**	(double dbu = 1)	Use of this method is deprecated

Detailed description

!=

Signature: *[const]* bool **!=** (const DCplxTrans other)

Description: Tests for inequality

*****

(1) Signature: *[const]* CplxTrans ***** (const CplxTrans t)

Description: Multiplication (concatenation) of transformations

t: The transformation to apply before

Returns: The modified transformation

The * operator returns self*t ("t is applied before this transformation").

(2) Signature: *[const]* double ***** (double d)

Description: Transforms a distance

d: The distance to transform

Returns: The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* DPoint ***** (const DPoint p)

Description: Transforms a point

p: The point to transform

Returns: The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* DVector ***** (const DVector p)

Description: Transforms a vector

v: The vector to transform

Returns: The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* DBox ***** (const DBox box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* DEdge ***** (const DEdge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* DPolygon ***** (const DPolygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform
Returns:	The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(8) Signature: *[const]* DPath ***** (const DPath path)

Description: Transforms a path

path:	The path to transform
Returns:	The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(9) Signature: *[const]* DText ***** (const DText text)

Description: Transforms a text

text:	The text to transform
Returns:	The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(10) Signature: *[const]* DCplxTrans ***** (const DCplxTrans t)

Description: Returns the concatenated transformation

t:	The transformation to apply before
Returns:	The modified transformation

The * operator returns self*t ("t is applied before this transformation").

<

Signature: *[const]* bool **<** (const DCplxTrans other)

Description: Provides a 'less' criterion for sorting

This method is provided to implement a sorting order. The definition of 'less' is opaque and might change in future versions.

==

Signature: *[const]* bool **==** (const DCplxTrans other)

Description: Tests for equality

M0

Signature: *[static]* DCplxTrans **M0**

Description: A constant giving "mirrored at the x-axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M0'. This is the getter.

M135

Signature: *[static]* DCplxTrans **M135**

Description: A constant giving "mirrored at the 135 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M135'. This is the getter.

M45

Signature: *[static]* DCplxTrans **M45**

Description: A constant giving "mirrored at the 45 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M45'. This is the getter.

M90

Signature: *[static]* DCplxTrans **M90**

Description: A constant giving "mirrored at the y (90 degree) axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M90'. This is the getter.

R0

Signature: *[static]* DCplxTrans **R0**

Description: A constant giving "unrotated" (unit) transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R0'. This is the getter.

R180

Signature: *[static]* DCplxTrans **R180**

Description: A constant giving "rotated by 180 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R180'. This is the getter.

R270

Signature: *[static]* DCplxTrans **R270**

Description: A constant giving "rotated by 270 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R270'. This is the getter.

R90

Signature: *[static]* DCplxTrans **R90**

Description: A constant giving "rotated by 90 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R90'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

angle

Signature: *[const]* double **angle**

Description: Gets the angle

Returns: The rotation angle this transformation provides in degree units (0..360 deg).

Note that the simple transformation returns the angle in units of 90 degree. Hence for a simple trans (i.e. Trans), a rotation angle of 180 degree delivers a value of 2 for the angle attribute. The complex transformation, supporting any rotation angle returns the angle in degree.

Python specific notes:
The object exposes a readable attribute 'angle'. This is the getter.

angle=

Signature: void **angle=** (double a)

Description: Sets the angle

a: The new angle

See angle for a description of that attribute.

Python specific notes:
The object exposes a writable attribute 'angle'. This is the setter.

assign

Signature: void **assign** (const DCplxTrans other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

ctrans

Signature: *[const]* double **ctrans** (double d)

Description: Transforms a distance

d: The distance to transform

Returns: The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disp

Signature: *[const]* DVector **disp**

Description: Gets the displacement

Python specific notes:
The object exposes a readable attribute 'disp'. This is the getter.

disp=

Signature: void **disp=** (const DVector u)

Description: Sets the displacement

u: The new displacement

Python specific notes:
The object exposes a writable attribute 'disp'. This is the setter.

dup

Signature: *[const]* new DCplxTrans ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_itrans

Signature: *[static]* new DCplxTrans ptr **from_itrans** (const CplxTrans trans, double dbu = 1)

Description: Creates a floating-point coordinate transformation from another coordinate flavour

Use of this method is deprecated. Use new instead

The 'dbu' argument is used to transform the input space from integer units to floating-point units. Formally, the DCplxTrans transformation is initialized with 'trans * to_dbu' where 'to_dbu' is the transformation into DBU space, or more precisely 'VCplxTrans(mag=1/dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new DCplxTrans ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given transformation. This method enables transformations as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

invert

Signature: DCplxTrans **invert**

Description: Inverts the transformation (in place)

Returns: The inverted transformation

Inverts the transformation and replaces this transformation by its inverted one.

inverted

Signature: *[const]* DCplxTrans **inverted**

Description: Returns the inverted transformation

Returns: The inverted transformation

Returns the inverted transformation. This method does not modify the transformation.

is_complex?

Signature: *[const]* bool **is_complex?**

Description: Returns true if the transformation is a complex one

If this predicate is false, the transformation can safely be converted to a simple transformation. Otherwise, this conversion will be lossy. The predicate value is equivalent to 'is_mag || !is_ortho'.

This method has been introduced in version 0.27.5.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_mag?

Signature: *[const]* bool **is_mag?**

Description: Tests, if the transformation is a magnifying one

This is the recommended test for checking if the transformation represents a magnification.

is_mirror?

Signature: *[const]* bool **is_mirror?**

Description: Gets the mirror flag

If this property is true, the transformation is composed of a mirroring at the x-axis followed by a rotation by the angle given by the angle property.

Python specific notes:

The object exposes a readable attribute 'mirror'. This is the getter.

is_ortho?

Signature: *[const]* bool **is_ortho?**

Description: Tests, if the transformation is an orthogonal transformation

If the rotation is by a multiple of 90 degree, this method will return true.

is_unity?

Signature: *[const]* bool **is_unity?**

Description: Tests, whether this is a unit transformation

mag

Signature: *[const]* double **mag**

Description: Gets the magnification

Python specific notes:
The object exposes a readable attribute 'mag'. This is the getter.

mag=

Signature: void **mag=** (double m)

Description: Sets the magnification

m: The new magnification

Python specific notes:
The object exposes a writable attribute 'mag'. This is the setter.

mirror=

Signature: void **mirror=** (bool m)

Description: Sets the mirror flag

m: The new mirror flag

"mirroring" describes a reflection at the x-axis which is included in the transformation prior to rotation.

Python specific notes:
The object exposes a writable attribute 'mirror'. This is the setter.

new

(1) Signature: *[static]* new DCplxTrans ptr **new** (const CplxTrans trans, double dbu = 1)

Description: Creates a floating-point coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the input space from integer units to floating-point units. Formally, the DCplxTrans transformation is initialized with 'trans * to_dbu' where 'to_dbu' is the transformation into DBU space, or more precisely 'VCplxTrans(mag=1/dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DCplxTrans ptr **new** (const ICplxTrans trans, double dbu = 1)

Description: Creates a floating-point coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the input and output space from integer units to floating-point units and vice versa. Formally, the DCplxTrans transformation is initialized with 'from_dbu * trans * to_dbu' where 'to_dbu' is the transformation into DBU space, or more precisely 'VCplxTrans(mag=1/dbu)'. 'from_dbu' is the transformation into micrometer space, or more precisely 'CplxTrans(mag=dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:

This method is the default initializer of the object.

(3) Signature: *[static]* new DCplxTrans ptr **new** (const VCplxTrans trans, double dbu = 1)

Description: Creates a floating-point coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the output space from integer units to floating-point units. Formally, the DCplxTrans transformation is initialized with 'from_dbu * trans' where 'from_dbu' is the transformation into micrometer space, or more precisely 'CplxTrans(mag=dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new DCplxTrans ptr **new**

Description: Creates a unit transformation

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new DCplxTrans ptr **new** (const DCplxTrans c, double mag = 1, const DVector u = 0,0)

Description: Creates a transformation from another transformation plus a magnification and displacement

c: The original transformation

u: The Additional displacement

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new DCplxTrans ptr **new** (const DCplxTrans c, double mag = 1, double x = 0, double y = 0)

Description: Creates a transformation from another transformation plus a magnification and displacement

c: The original transformation

x: The Additional displacement (x)

y: The Additional displacement (y)

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new DCplxTrans ptr **new** (double x, double y)

Description: Creates a transformation from a x and y displacement

x: The x displacement

y:	The y displacement

This constructor will create a transformation with the specified displacement but no rotation.

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new DCplxTrans ptr **new** (const DTrans t, double mag = 1)

Description: Creates a transformation from a simple transformation and a magnification

Creates a magnifying transformation from a simple transformation and a magnification.

Python specific notes:
This method is the default initializer of the object.

(9) Signature: *[static]* new DCplxTrans ptr **new** (const DVector u)

Description: Creates a transformation from a displacement

Creates a transformation with a displacement only.

This method has been added in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(10) Signature: *[static]* new DCplxTrans ptr **new** (double mag = 1, double rot = 0, bool mirrx = false, const DVector u = 0,0)

Description: Creates a transformation using magnification, angle, mirror flag and displacement

mag:	The magnification
rot:	The rotation angle in units of degree
mirrx:	True, if mirrored at x axis
u:	The displacement

The sequence of operations is: magnification, mirroring at x axis, rotation, application of displacement.

Python specific notes:
This method is the default initializer of the object.

(11) Signature: *[static]* new DCplxTrans ptr **new** (double mag = 1, double rot = 0, bool mirrx = false, double x = 0, double y = 0)

Description: Creates a transformation using magnification, angle, mirror flag and displacement

mag:	The magnification
rot:	The rotation angle in units of degree
mirrx:	True, if mirrored at x axis
x:	The x displacement
y:	The y displacement

The sequence of operations is: magnification, mirroring at x axis, rotation, application of displacement.

Python specific notes:
This method is the default initializer of the object.

rot

Signature: *[const]* int **rot**

Description: Returns the respective simple transformation equivalent rotation code if possible

If this transformation is orthogonal (is_ortho () == true), then this method will return the corresponding fixpoint transformation, not taking into account magnification and displacement. If the transformation is not orthogonal, the result reflects the quadrant the rotation goes into.

s_trans

Signature: *[const]* DTrans **s_trans**

Description: Extracts the simple transformation part

The simple transformation part does not reflect magnification or arbitrary angles. Rotation angles are rounded down to multiples of 90 degree. Magnification is fixed to 1.0.

to_itrans

Signature: *[const]* ICplxTrans **to_itrans** (double dbu = 1)

Description: Converts the transformation to another transformation with integer input and output coordinates

Use of this method is deprecated

The database unit can be specified to translate the floating-point coordinate displacement in micron units to an integer-coordinate displacement in database units. The displacement's' coordinates will be divided by the database unit.

This method is redundant with the conversion constructors. Instead of 'to_itrans' use the conversion constructor:

```
itrans = RBA::ICplxTrans::new(dtrans, dbu)
```

This method has been introduced in version 0.25 and was deprecated in version 0.29.

to_s

Signature: *[const]* string **to_s** (bool lazy = false, double dbu = 0)

Description: String conversion

If 'lazy' is true, some parts are omitted when not required. If a DBU is given, the output units will be micrometers.

The lazy and DBU arguments have been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

to_trans

Signature: *[const]* CplxTrans **to_trans** (double dbu = 1)

Description: Converts the transformation to another transformation with integer input coordinates

Use of this method is deprecated

This method is redundant with the conversion constructors. Instead of 'to_trans' use the conversion constructor:

```
trans = RBA::CplxTrans::new(dtrans, dbu)
```

This method has been introduced in version 0.25 and was deprecated in version 0.29.

to_vtrans

Signature: *[const]* VCplxTrans **to_vtrans** (double dbu = 1)

Description: Converts the transformation to another transformation with integer output coordinates

Use of this method is deprecated

The database unit can be specified to translate the floating-point coordinate displacement in micron units to an integer-coordinate displacement in database units. The displacement's' coordinates will be divided by the database unit.

This method is redundant with the conversion constructors. Instead of 'to_vtrans' use the conversion constructor:

```
vtrans = RBA::VCplxTrans::new(dtrans, dbu)
```

This method has been introduced in version 0.25 and was deprecated in version 0.29.

trans

(1) Signature: *[const]* DPoint **trans** (const DPoint p)

Description: Transforms a point

p:	The point to transform
Returns:	The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* DVector **trans** (const DVector p)

Description: Transforms a vector

v:	The vector to transform
Returns:	The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* DBox **trans** (const DBox box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* DEdge **trans** (const DEdge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:

This method also implements '__rmul__'.

(5) Signature: *[const]* DPolygon **trans** (const DPolygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform
Returns:	The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* DPath **trans** (const DPath path)

Description: Transforms a path

path:	The path to transform
Returns:	The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* DText **trans** (const DText text)

Description: Transforms a text

text:	The text to transform
Returns:	The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

4.108. API reference - Class CplxTrans

Notation used in Ruby API documentation

Module: db

Description: A complex transformation

A complex transformation provides magnification, mirroring at the x-axis, rotation by an arbitrary angle and a displacement. This is also the order, the operations are applied. This version can transform integer-coordinate objects into floating-point coordinate objects. This is the generic and exact case, for example for non-integer magnifications.

Complex transformations are extensions of the simple transformation classes (Trans or DTrans in that case) and behave similar.

Transformations can be used to transform points or other objects. Transformations can be combined with the '*' operator to form the transformation which is equivalent to applying the second and then the first. Here is some code:

```
# Create a transformation that applies a magnification of 1.5, a rotation by 90 degree
# and displacement of 10 in x and 20 units in y direction:
t = RBA::DCplxTrans::new(1.5, 90, false, 10.0, 20.0)
t.to_s            # r90 *1.5 10,20
# compute the inverse:
t.inverted.to_s   # r270 *0.666666667 -13,7
# Combine with another displacement (applied after that):
(RBA::DCplxTrans::new(5, 5) * t).to_s    # r90 *1.5 15,25
# Transform a point:
t.trans(RBA::DPoint::new(100, 200)).to_s # -290,170
```

The inverse type of the CplxTrans type is VCplxTrans which will transform floating-point to integer coordinate objects. Transformations of CplxTrans type can be concatenated (operator *) with either itself or with transformations of compatible input or output type. This means, the operator CplxTrans * ICplxTrans is allowed (output types of ICplxTrans and input of CplxTrans are identical) while CplxTrans * DCplxTrans is not. See The Database API for more details about the database objects.

Public constructors

new CplxTrans ptr	**new**	(const DCplxTrans trans, double dbu = 1)	Creates an integer-to-floating-point coordinate transformation from another coordinate flavour
new CplxTrans ptr	**new**	(const ICplxTrans trans, double dbu = 1)	Creates an integer-to-floating-point coordinate transformation from another coordinate flavour
new CplxTrans ptr	**new**	(const VCplxTrans trans, double dbu = 1)	Creates an integer-to-floating-point coordinate transformation from another coordinate flavour
new CplxTrans ptr	**new**		Creates a unit transformation
new CplxTrans ptr	**new**	(const CplxTrans c, double mag = 1, const DVector u = 0,0)	Creates a transformation from another transformation plus a magnification and displacement
new CplxTrans ptr	**new**	(const CplxTrans c, double mag = 1, double x = 0, double y = 0)	Creates a transformation from another transformation plus a magnification and displacement
new CplxTrans ptr	**new**	(double x, double y)	Creates a transformation from a x and y displacement
new CplxTrans ptr	**new**	(const Trans t, double mag = 1)	Creates a transformation from a simple transformation and a magnification

new CplxTrans ptr	new	(const DVector u)	Creates a transformation from a displacement
new CplxTrans ptr	new	(double mag = 1, double rot = 0, bool mirrx = false, const DVector u = 0,0)	Creates a transformation using magnification, angle, mirror flag and displacement
new CplxTrans ptr	new	(double mag = 1, double rot = 0, bool mirrx = false, double x = 0, double y = 0)	Creates a transformation using magnification, angle, mirror flag and displacement

Public methods

[const]	bool	!=	(const CplxTrans other)	Tests for inequality
[const]	DCplxTrans	*	(const VCplxTrans t)	Multiplication (concatenation) of transformations
[const]	CplxTrans	*	(const ICplxTrans t)	Multiplication (concatenation) of transformations
[const]	double	*	(unsigned int d)	Transforms a distance
[const]	DPoint	*	(const Point p)	Transforms a point
[const]	DVector	*	(const Vector p)	Transforms a vector
[const]	DBox	*	(const Box box)	Transforms a box
[const]	DEdge	*	(const Edge edge)	Transforms an edge
[const]	DPolygon	*	(const Polygon polygon)	Transforms a polygon
[const]	DPath	*	(const Path path)	Transforms a path
[const]	DText	*	(const Text text)	Transforms a text
[const]	CplxTrans	*	(const CplxTrans t)	Returns the concatenated transformation
[const]	bool	<	(const CplxTrans other)	Provides a 'less' criterion for sorting
[const]	bool	==	(const CplxTrans other)	Tests for equality
	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed

[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	double	**angle**		Gets the angle
	void	**angle=**	(double a)	Sets the angle
	void	**assign**	(const CplxTrans other)	Assigns another object to self
[const]	double	**ctrans**	(unsigned int d)	Transforms a distance
[const]	DVector	**disp**		Gets the displacement
	void	**disp=**	(const DVector u)	Sets the displacement
[const]	new CplxTrans ptr	**dup**		Creates a copy of self
[const]	unsigned long	**hash**		Computes a hash value
	CplxTrans	**invert**		Inverts the transformation (in place)
[const]	VCplxTrans	**inverted**		Returns the inverted transformation
[const]	bool	**is_complex?**		Returns true if the transformation is a complex one
[const]	bool	**is_mag?**		Tests, if the transformation is a magnifying one
[const]	bool	**is_mirror?**		Gets the mirror flag
[const]	bool	**is_ortho?**		Tests, if the transformation is an orthogonal transformation
[const]	bool	**is_unity?**		Tests, whether this is a unit transformation
[const]	double	**mag**		Gets the magnification
	void	**mag=**	(double m)	Sets the magnification
	void	**mirror=**	(bool m)	Sets the mirror flag
[const]	int	**rot**		Returns the respective simple transformation equivalent rotation code if possible
[const]	Trans	**s_trans**		Extracts the simple transformation part
[const]	string	**to_s**	(bool lazy = false, double dbu = 0)	String conversion

[const]	DPoint	**trans**	(const Point p)	Transforms a point
[const]	DVector	**trans**	(const Vector p)	Transforms a vector
[const]	DBox	**trans**	(const Box box)	Transforms a box
[const]	DEdge	**trans**	(const Edge edge)	Transforms an edge
[const]	DPolygon	**trans**	(const Polygon polygon)	Transforms a polygon
[const]	DPath	**trans**	(const Path path)	Transforms a path
[const]	DText	**trans**	(const Text text)	Transforms a text

Public static methods and constants

	CplxTrans	**M0**		A constant giving "mirrored at the x-axis" transformation
	CplxTrans	**M135**		A constant giving "mirrored at the 135 degree axis" transformation
	CplxTrans	**M45**		A constant giving "mirrored at the 45 degree axis" transformation
	CplxTrans	**M90**		A constant giving "mirrored at the y (90 degree) axis" transformation
	CplxTrans	**R0**		A constant giving "unrotated" (unit) transformation
	CplxTrans	**R180**		A constant giving "rotated by 180 degree counterclockwise" transformation
	CplxTrans	**R270**		A constant giving "rotated by 270 degree counterclockwise" transformation
	CplxTrans	**R90**		A constant giving "rotated by 90 degree counterclockwise" transformation
	new CplxTrans ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new CplxTrans ptr	**from_dtrans**	(const DCplxTrans trans,	Use of this method is deprecated. Use new instead

			double dbu = 1)	
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	ICplxTrans	**to_itrans**	(double dbu = 1)	Use of this method is deprecated
[const]	DCplxTrans	**to_trans**	(double dbu = 1)	Use of this method is deprecated
[const]	VCplxTrans	**to_vtrans**	(double dbu = 1)	Use of this method is deprecated

Detailed description

!=

Signature: *[const]* bool **!=** (const CplxTrans other)

Description: Tests for inequality

*****

(1) Signature: *[const]* DCplxTrans ***** (const VCplxTrans t)

Description: Multiplication (concatenation) of transformations

t: The transformation to apply before

Returns: The modified transformation

The * operator returns self*t ("t is applied before this transformation").

(2) Signature: *[const]* CplxTrans ***** (const ICplxTrans t)

Description: Multiplication (concatenation) of transformations

t: The transformation to apply before

Returns: The modified transformation

The * operator returns self*t ("t is applied before this transformation").

(3) Signature: *[const]* double ***** (unsigned int d)

Description: Transforms a distance

d: The distance to transform

Returns: The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* DPoint ***** (const Point p)

Description: Transforms a point

p: The point to transform

Returns: The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:

This method also implements '__rmul__'.

(5) Signature: *[const]* DVector ***** (const Vector p)

Description: Transforms a vector

v:	The vector to transform
Returns:	The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* DBox ***** (const Box box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* DEdge ***** (const Edge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(8) Signature: *[const]* DPolygon ***** (const Polygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform
Returns:	The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(9) Signature: *[const]* DPath ***** (const Path path)

Description: Transforms a path

path:	The path to transform
Returns:	The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(10) Signature: *[const]* DText ***** (const Text text)

Description: Transforms a text

text:	The text to transform
Returns:	The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(11) Signature: *[const]* CplxTrans ***** (const CplxTrans t)

Description: Returns the concatenated transformation

t:	The transformation to apply before
Returns:	The modified transformation

The * operator returns self*t ("t is applied before this transformation").

<

Signature: *[const]* bool **<** (const CplxTrans other)

Description: Provides a 'less' criterion for sorting

This method is provided to implement a sorting order. The definition of 'less' is opaque and might change in future versions.

==

Signature: *[const]* bool **==** (const CplxTrans other)

Description: Tests for equality

M0

Signature: *[static]* CplxTrans **M0**

Description: A constant giving "mirrored at the x-axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M0'. This is the getter.

M135

Signature: *[static]* CplxTrans **M135**

Description: A constant giving "mirrored at the 135 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M135'. This is the getter.

M45

Signature: *[static]* CplxTrans **M45**

Description: A constant giving "mirrored at the 45 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M45'. This is the getter.

M90

Signature: *[static]* CplxTrans **M90**

Description: A constant giving "mirrored at the y (90 degree) axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M90'. This is the getter.

R0

Signature: *[static]* CplxTrans **R0**

Description: A constant giving "unrotated" (unit) transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R0'. This is the getter.

R180

Signature: *[static]* CplxTrans **R180**

Description: A constant giving "rotated by 180 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R180'. This is the getter.

R270

Signature: *[static]* CplxTrans **R270**

Description: A constant giving "rotated by 270 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R270'. This is the getter.

R90

Signature: *[static]* CplxTrans **R90**

Description: A constant giving "rotated by 90 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R90'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

angle

Signature: *[const]* double **angle**

Description: Gets the angle

Returns: The rotation angle this transformation provides in degree units (0..360 deg).

Note that the simple transformation returns the angle in units of 90 degree. Hence for a simple trans (i.e. Trans), a rotation angle of 180 degree delivers a value of 2 for the angle attribute. The complex transformation, supporting any rotation angle returns the angle in degree.

Python specific notes:
The object exposes a readable attribute 'angle'. This is the getter.

angle=

Signature: void **angle=** (double a)

Description: Sets the angle

a: The new angle

See angle for a description of that attribute.

Python specific notes:
The object exposes a writable attribute 'angle'. This is the setter.

assign

Signature: void **assign** (const CplxTrans other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

ctrans

Signature: *[const]* double **ctrans** (unsigned int d)

Description: Transforms a distance

d: The distance to transform

Returns: The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disp

Signature: *[const]* DVector **disp**

Description: Gets the displacement

Python specific notes:
The object exposes a readable attribute 'disp'. This is the getter.

disp=

Signature: void **disp=** (const DVector u)

Description: Sets the displacement

u: The new displacement

Python specific notes:
The object exposes a writable attribute 'disp'. This is the setter.

dup

Signature: *[const]* new CplxTrans ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_dtrans

Signature: *[static]* new CplxTrans ptr **from_dtrans** (const DCplxTrans trans, double dbu = 1)

Description: Creates an integer-to-floating-point coordinate transformation from another coordinate flavour

Use of this method is deprecated. Use new instead

The 'dbu' argument is used to transform the input space from floating-point units to integer units. Formally, the CplxTrans transformation is initialized with 'trans * from_dbu' where 'from_dbu' is the transformation into micrometer space, or more precisely 'CplxTrans(mag=dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new CplxTrans ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given transformation. This method enables transformations as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

invert

Signature: CplxTrans **invert**

Description: Inverts the transformation (in place)

Returns: The inverted transformation

Inverts the transformation and replaces this transformation by its inverted one.

inverted

Signature: *[const]* VCplxTrans **inverted**

Description: Returns the inverted transformation

Returns: The inverted transformation

Returns the inverted transformation. This method does not modify the transformation.

is_complex?

Signature: *[const]* bool **is_complex?**

Description: Returns true if the transformation is a complex one

If this predicate is false, the transformation can safely be converted to a simple transformation. Otherwise, this conversion will be lossy. The predicate value is equivalent to 'is_mag || !is_ortho'.

This method has been introduced in version 0.27.5.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_mag?

Signature: *[const]* bool **is_mag?**

Description: Tests, if the transformation is a magnifying one

This is the recommended test for checking if the transformation represents a magnification.

is_mirror?

Signature: *[const]* bool **is_mirror?**

Description: Gets the mirror flag

If this property is true, the transformation is composed of a mirroring at the x-axis followed by a rotation by the angle given by the angle property.

Python specific notes:
The object exposes a readable attribute 'mirror'. This is the getter.

is_ortho?

Signature: *[const]* bool **is_ortho?**

Description: Tests, if the transformation is an orthogonal transformation

If the rotation is by a multiple of 90 degree, this method will return true.

is_unity?

Signature: *[const]* bool **is_unity?**

Description: Tests, whether this is a unit transformation

mag

Signature: *[const]* double **mag**

Description: Gets the magnification

Python specific notes:
The object exposes a readable attribute 'mag'. This is the getter.

mag=

Signature: void **mag=** (double m)

Description: Sets the magnification

m: The new magnification

Python specific notes:
The object exposes a writable attribute 'mag'. This is the setter.

mirror=

Signature: void **mirror=** (bool m)

Description: Sets the mirror flag

m: The new mirror flag

"mirroring" describes a reflection at the x-axis which is included in the transformation prior to rotation.

Python specific notes:
The object exposes a writable attribute 'mirror'. This is the setter.

new

(1) Signature: *[static]* new CplxTrans ptr **new** (const DCplxTrans trans, double dbu = 1)

Description: Creates an integer-to-floating-point coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the input space from floating-point units to integer units. Formally, the CplxTrans transformation is initialized with 'trans * from_dbu' where 'from_dbu' is the transformation into micrometer space, or more precisely 'CplxTrans(mag=dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new CplxTrans ptr **new** (const ICplxTrans trans, double dbu = 1)

Description: Creates an integer-to-floating-point coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the output space from integer units to floating-point units. Formally, the CplxTrans transformation is initialized with 'from_dbu * trans' where 'from_dbu' is the transformation into micrometer space, or more precisely 'CplxTrans(mag=dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new CplxTrans ptr **new** (const VCplxTrans trans, double dbu = 1)

Description: Creates an integer-to-floating-point coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the input and output space from integer units to floating-point units and vice versa. Formally, the DCplxTrans transformation is initialized with 'from_dbu * trans * from_dbu' where 'from_dbu' is the transformation into micrometer space, or more precisely 'CplxTrans(mag=dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new CplxTrans ptr **new**

Description: Creates a unit transformation

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new CplxTrans ptr **new** (const CplxTrans c, double mag = 1, const DVector u = 0,0)

Description: Creates a transformation from another transformation plus a magnification and displacement

c:	The original transformation
u:	The Additional displacement

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new CplxTrans ptr **new** (const CplxTrans c, double mag = 1, double x = 0, double y = 0)

Description: Creates a transformation from another transformation plus a magnification and displacement

c:	The original transformation
x:	The Additional displacement (x)
y:	The Additional displacement (y)

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new CplxTrans ptr **new** (double x, double y)

Description: Creates a transformation from a x and y displacement

x:	The x displacement
y:	The y displacement

This constructor will create a transformation with the specified displacement but no rotation.

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new CplxTrans ptr **new** (const Trans t, double mag = 1)

Description: Creates a transformation from a simple transformation and a magnification

Creates a magnifying transformation from a simple transformation and a magnification.

Python specific notes:
This method is the default initializer of the object.

(9) Signature: *[static]* new CplxTrans ptr **new** (const DVector u)

Description: Creates a transformation from a displacement

Creates a transformation with a displacement only.

This method has been added in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(10) Signature: *[static]* new CplxTrans ptr **new** (double mag = 1, double rot = 0, bool mirrx = false, const DVector u = 0,0)

Description: Creates a transformation using magnification, angle, mirror flag and displacement

mag:	The magnification
rot:	The rotation angle in units of degree
mirrx:	True, if mirrored at x axis
u:	The displacement

The sequence of operations is: magnification, mirroring at x axis, rotation, application of displacement.

Python specific notes:
This method is the default initializer of the object.

(11) Signature: *[static]* new CplxTrans ptr **new** (double mag = 1, double rot = 0, bool mirrx = false, double x = 0, double y = 0)

Description: Creates a transformation using magnification, angle, mirror flag and displacement

mag:	The magnification
rot:	The rotation angle in units of degree
mirrx:	True, if mirrored at x axis

x: The x displacement

y: The y displacement

The sequence of operations is: magnification, mirroring at x axis, rotation, application of displacement.

Python specific notes:
This method is the default initializer of the object.

rot

Signature: *[const]* int **rot**

Description: Returns the respective simple transformation equivalent rotation code if possible

If this transformation is orthogonal (is_ortho () == true), then this method will return the corresponding fixpoint transformation, not taking into account magnification and displacement. If the transformation is not orthogonal, the result reflects the quadrant the rotation goes into.

s_trans

Signature: *[const]* Trans **s_trans**

Description: Extracts the simple transformation part

The simple transformation part does not reflect magnification or arbitrary angles. Rotation angles are rounded down to multiples of 90 degree. Magnification is fixed to 1.0.

to_itrans

Signature: *[const]* ICplxTrans **to_itrans** (double dbu = 1)

Description: Converts the transformation to another transformation with integer input and output coordinates

Use of this method is deprecated

This method is redundant with the conversion constructors. Instead of 'to_itrans' use the conversion constructor:

```
itrans = RBA::ICplxTrans::new(trans, dbu)
```

This method has been introduced in version 0.25 and was deprecated in version 0.29.

to_s

Signature: *[const]* string **to_s** (bool lazy = false, double dbu = 0)

Description: String conversion

If 'lazy' is true, some parts are omitted when not required. If a DBU is given, the output units will be micrometers.

The lazy and DBU arguments have been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

to_trans

Signature: *[const]* DCplxTrans **to_trans** (double dbu = 1)

Description: Converts the transformation to another transformation with floating-point input coordinates

Use of this method is deprecated

This method is redundant with the conversion constructors. Instead of 'to_trans' use the conversion constructor:

```
dtrans = RBA::DCplxTrans::new(trans, dbu)
```

This method has been introduced in version 0.25 and was deprecated in version 0.29.

to_vtrans

Signature: *[const]* VCplxTrans **to_vtrans** (double dbu = 1)

Description: Converts the transformation to another transformation with integer output and floating-point input coordinates

Use of this method is deprecated

This method is redundant with the conversion constructors. Instead of 'to_vtrans' use the conversion constructor:

```
vtrans = RBA::VCplxTrans::new(trans, dbu)
```

This method has been introduced in version 0.25 and was deprecated in version 0.29.

trans

(1) Signature: *[const]* DPoint **trans** (const Point p)

Description: Transforms a point

p:	The point to transform
Returns:	The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* DVector **trans** (const Vector p)

Description: Transforms a vector

v:	The vector to transform
Returns:	The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* DBox **trans** (const Box box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* DEdge **trans** (const Edge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* DPolygon **trans** (const Polygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform
Returns:	The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* DPath **trans** (const Path path)

Description: Transforms a path

path:	The path to transform
Returns:	The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* DText **trans** (const Text text)

Description: Transforms a text

text:	The text to transform
Returns:	The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

4.109. API reference - Class ICplxTrans

Notation used in Ruby API documentation

Module: db

Description: A complex transformation

A complex transformation provides magnification, mirroring at the x-axis, rotation by an arbitrary angle and a displacement. This is also the order, the operations are applied. This version can transform integer-coordinate objects into the same, which may involve rounding and can be inexact.

Complex transformations are extensions of the simple transformation classes (Trans in that case) and behave similar.

Transformations can be used to transform points or other objects. Transformations can be combined with the '*' operator to form the transformation which is equivalent to applying the second and then the first. Here is some code:

```
# Create a transformation that applies a magnification of 1.5, a rotation by 90 degree
# and displacement of 10 in x and 20 units in y direction:
t = RBA::ICplxTrans::new(1.5, 90, false, 10.0, 20.0)
t.to_s            # r90 *1.5 10,20
# compute the inverse:
t.inverted.to_s   # r270 *0.666666667 -13,7
# Combine with another displacement (applied after that):
(RBA::ICplxTrans::new(5, 5) * t).to_s    # r90 *1.5 15,25
# Transform a point:
t.trans(RBA::Point::new(100, 200)).to_s  # -290,170
```

This class has been introduced in version 0.18.

See The Database API for more details about the database objects.

Public constructors

new ICplxTrans ptr	**new**	(const DCplxTrans trans, double dbu = 1)	Creates an integer coordinate transformation from another coordinate flavour
new ICplxTrans ptr	**new**	(const CplxTrans trans, double dbu = 1)	Creates an integer coordinate transformation from another coordinate flavour
new ICplxTrans ptr	**new**	(const VCplxTrans trans, double dbu = 1)	Creates an integer coordinate transformation from another coordinate flavour
new ICplxTrans ptr	**new**		Creates a unit transformation
new ICplxTrans ptr	**new**	(const ICplxTrans c, double mag = 1, const Vector u = 0,0)	Creates a transformation from another transformation plus a magnification and displacement
new ICplxTrans ptr	**new**	(const ICplxTrans c, double mag = 1, int x = 0, int y = 0)	Creates a transformation from another transformation plus a magnification and displacement
new ICplxTrans ptr	**new**	(int x, int y)	Creates a transformation from a x and y displacement
new ICplxTrans ptr	**new**	(const Trans t, double mag = 1)	Creates a transformation from a simple transformation and a magnification

new ICplxTrans ptr	new	(const Vector u)	Creates a transformation from a displacement
new ICplxTrans ptr	new	(double mag = 1, double rot = 0, bool mirrx = false, const Vector u = 0,0)	Creates a transformation using magnification, angle, mirror flag and displacement
new ICplxTrans ptr	new	(double mag = 1, double rot = 0, bool mirrx = false, int x = 0, int y = 0)	Creates a transformation using magnification, angle, mirror flag and displacement

Public methods

[const]	bool	!=	(const ICplxTrans other)	Tests for inequality
[const]	VCplxTrans	*	(const VCplxTrans t)	Multiplication (concatenation) of transformations
[const]	unsigned int	*	(unsigned int d)	Transforms a distance
[const]	Point	*	(const Point p)	Transforms a point
[const]	Vector	*	(const Vector p)	Transforms a vector
[const]	Box	*	(const Box box)	Transforms a box
[const]	Edge	*	(const Edge edge)	Transforms an edge
[const]	Polygon	*	(const Polygon polygon)	Transforms a polygon
[const]	Path	*	(const Path path)	Transforms a path
[const]	Text	*	(const Text text)	Transforms a text
[const]	ICplxTrans	*	(const ICplxTrans t)	Returns the concatenated transformation
[const]	bool	<	(const ICplxTrans other)	Provides a 'less' criterion for sorting
[const]	bool	==	(const ICplxTrans other)	Tests for equality
	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference

	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	double	angle		Gets the angle
	void	angle=	(double a)	Sets the angle
	void	assign	(const ICplxTrans other)	Assigns another object to self
[const]	unsigned int	ctrans	(unsigned int d)	Transforms a distance
[const]	Vector	disp		Gets the displacement
	void	disp=	(const Vector u)	Sets the displacement
[const]	new ICplxTrans ptr	dup		Creates a copy of self
[const]	unsigned long	hash		Computes a hash value
	ICplxTrans	invert		Inverts the transformation (in place)
[const]	ICplxTrans	inverted		Returns the inverted transformation
[const]	bool	is_complex?		Returns true if the transformation is a complex one
[const]	bool	is_mag?		Tests, if the transformation is a magnifying one
[const]	bool	is_mirror?		Gets the mirror flag
[const]	bool	is_ortho?		Tests, if the transformation is an orthogonal transformation
[const]	bool	is_unity?		Tests, whether this is a unit transformation
[const]	double	mag		Gets the magnification
	void	mag=	(double m)	Sets the magnification
	void	mirror=	(bool m)	Sets the mirror flag
[const]	int	rot		Returns the respective simple transformation equivalent rotation code if possible
[const]	Trans	s_trans		Extracts the simple transformation part
[const]	string	to_s	(bool lazy = false, double dbu = 0)	String conversion
[const]	Point	trans	(const Point p)	Transforms a point

[const]	Vector	**trans**	(const Vector p)	Transforms a vector
[const]	Box	**trans**	(const Box box)	Transforms a box
[const]	Edge	**trans**	(const Edge edge)	Transforms an edge
[const]	Polygon	**trans**	(const Polygon polygon)	Transforms a polygon
[const]	Path	**trans**	(const Path path)	Transforms a path
[const]	Text	**trans**	(const Text text)	Transforms a text

Public static methods and constants

	ICplxTrans	**M0**		A constant giving "mirrored at the x-axis" transformation
	ICplxTrans	**M135**		A constant giving "mirrored at the 135 degree axis" transformation
	ICplxTrans	**M45**		A constant giving "mirrored at the 45 degree axis" transformation
	ICplxTrans	**M90**		A constant giving "mirrored at the y (90 degree) axis" transformation
	ICplxTrans	**R0**		A constant giving "unrotated" (unit) transformation
	ICplxTrans	**R180**		A constant giving "rotated by 180 degree counterclockwise" transformation
	ICplxTrans	**R270**		A constant giving "rotated by 270 degree counterclockwise" transformation
	ICplxTrans	**R90**		A constant giving "rotated by 90 degree counterclockwise" transformation
	new ICplxTrans ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	new ICplxTrans ptr	**from_dtrans**	(const DCplxTrans trans, double dbu = 1)	Use of this method is deprecated. Use new instead
[static]	new ICplxTrans ptr	**from_trans**	(const CplxTrans trans,	Use of this method is deprecated. Use new instead

			double dbu = 1)	
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	DCplxTrans	**to_itrans**	(double dbu = 1)	Use of this method is deprecated
[const]	VCplxTrans	**to_trans**	(double dbu = 1)	Use of this method is deprecated
[const]	CplxTrans	**to_vtrans**	(double dbu = 1)	Use of this method is deprecated

Detailed description

!=

Signature: *[const]* bool **!=** (const ICplxTrans other)

Description: Tests for inequality

*****

(1) Signature: *[const]* VCplxTrans ***** (const VCplxTrans t)

Description: Multiplication (concatenation) of transformations

t: The transformation to apply before

Returns: The modified transformation

The * operator returns self*t ("t is applied before this transformation").

(2) Signature: *[const]* unsigned int ***** (unsigned int d)

Description: Transforms a distance

d: The distance to transform

Returns: The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* Point ***** (const Point p)

Description: Transforms a point

p: The point to transform

Returns: The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* Vector ***** (const Vector p)

Description: Transforms a vector

v: The vector to transform

Returns: The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* Box * (const Box box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* Edge * (const Edge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* Polygon * (const Polygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform
Returns:	The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(8) Signature: *[const]* Path * (const Path path)

Description: Transforms a path

path:	The path to transform
Returns:	The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(9) Signature: *[const]* Text * (const Text text)

Description: Transforms a text

text:	The text to transform
Returns:	The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(10) Signature: *[const]* ICplxTrans ***** (const ICplxTrans t)

Description: Returns the concatenated transformation

t:	The transformation to apply before
Returns:	The modified transformation

The * operator returns self*t ("t is applied before this transformation").

<

Signature: *[const]* bool **<** (const ICplxTrans other)

Description: Provides a 'less' criterion for sorting

This method is provided to implement a sorting order. The definition of 'less' is opaque and might change in future versions.

==

Signature: *[const]* bool **==** (const ICplxTrans other)

Description: Tests for equality

M0

Signature: *[static]* ICplxTrans **M0**

Description: A constant giving "mirrored at the x-axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M0'. This is the getter.

M135

Signature: *[static]* ICplxTrans **M135**

Description: A constant giving "mirrored at the 135 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M135'. This is the getter.

M45

Signature: *[static]* ICplxTrans **M45**

Description: A constant giving "mirrored at the 45 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M45'. This is the getter.

M90

Signature: *[static]* ICplxTrans **M90**

Description: A constant giving "mirrored at the y (90 degree) axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M90'. This is the getter.

R0

Signature: *[static]* ICplxTrans **R0**

Description: A constant giving "unrotated" (unit) transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R0'. This is the getter.

R180

Signature: *[static]* ICplxTrans **R180**

Description: A constant giving "rotated by 180 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R180'. This is the getter.

R270

Signature: *[static]* ICplxTrans **R270**

Description: A constant giving "rotated by 270 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R270'. This is the getter.

R90

Signature: *[static]* ICplxTrans **R90**

Description: A constant giving "rotated by 90 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R90'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

angle

Signature: *[const]* double **angle**

Description: Gets the angle

Returns: The rotation angle this transformation provides in degree units (0..360 deg).

Note that the simple transformation returns the angle in units of 90 degree. Hence for a simple trans (i.e. Trans), a rotation angle of 180 degree delivers a value of 2 for the angle attribute. The complex transformation, supporting any rotation angle returns the angle in degree.

Python specific notes:
The object exposes a readable attribute 'angle'. This is the getter.

angle=

Signature: void **angle=** (double a)

Description: Sets the angle

a: The new angle

See angle for a description of that attribute.

Python specific notes:
The object exposes a writable attribute 'angle'. This is the setter.

assign

Signature: void **assign** (const ICplxTrans other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

ctrans

Signature: *[const]* unsigned int **ctrans** (unsigned int d)

Description: Transforms a distance

d: The distance to transform

Returns: The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disp

Signature: *[const]* Vector **disp**

Description: Gets the displacement

Python specific notes:
The object exposes a readable attribute 'disp'. This is the getter.

disp=

Signature: void **disp=** (const Vector u)

Description: Sets the displacement

u: The new displacement

Python specific notes:
The object exposes a writable attribute 'disp'. This is the setter.

dup

Signature: *[const]* new ICplxTrans ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_dtrans

Signature: *[static]* new ICplxTrans ptr **from_dtrans** (const DCplxTrans trans, double dbu = 1)

Description: Creates an integer coordinate transformation from another coordinate flavour

Use of this method is deprecated. Use new instead

The 'dbu' argument is used to transform the input space and output space from floating-point units to integer units and vice versa. Formally, the ICplxTrans transformation is initialized with 'to_dbu * trans * from_dbu' where 'from_dbu' is the transformation into micrometer space, or more precisely 'CplxTrans(mag=dbu)' and 'to_dbu' is the transformation into DBU space, or more precisely 'VCplxTrans(mag=1/dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

from_s

Signature: *[static]* new ICplxTrans ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

from_trans

Signature: *[static]* new ICplxTrans ptr **from_trans** (const CplxTrans trans, double dbu = 1)

Description: Creates an integer coordinate transformation from another coordinate flavour

Use of this method is deprecated. Use new instead

The 'dbu' argument is used to transform the output space from floating-point units to integer units. Formally, the CplxTrans transformation is initialized with 'to_dbu * trans' where 'to_dbu' is the transformation into DBU space, or more precisely 'VCplxTrans(mag=1/dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given transformation. This method enables transformations as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

invert

Signature: ICplxTrans **invert**

Description: Inverts the transformation (in place)

Returns: The inverted transformation

Inverts the transformation and replaces this transformation by its inverted one.

inverted

Signature: *[const]* ICplxTrans **inverted**

Description: Returns the inverted transformation

Returns: The inverted transformation

Returns the inverted transformation. This method does not modify the transformation.

is_complex?

Signature: *[const]* bool **is_complex?**

Description: Returns true if the transformation is a complex one

If this predicate is false, the transformation can safely be converted to a simple transformation. Otherwise, this conversion will be lossy. The predicate value is equivalent to 'is_mag || !is_ortho'.

This method has been introduced in version 0.27.5.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_mag?

Signature: *[const]* bool **is_mag?**

Description: Tests, if the transformation is a magnifying one

This is the recommended test for checking if the transformation represents a magnification.

is_mirror?

Signature: *[const]* bool **is_mirror?**

Description: Gets the mirror flag

If this property is true, the transformation is composed of a mirroring at the x-axis followed by a rotation by the angle given by the angle property.

Python specific notes:
The object exposes a readable attribute 'mirror'. This is the getter.

is_ortho?

Signature: *[const]* bool **is_ortho?**

Description: Tests, if the transformation is an orthogonal transformation

If the rotation is by a multiple of 90 degree, this method will return true.

is_unity?

Signature: *[const]* bool **is_unity?**

Description: Tests, whether this is a unit transformation

mag

Signature: *[const]* double **mag**

Description: Gets the magnification

Python specific notes:
The object exposes a readable attribute 'mag'. This is the getter.

mag=

Signature: void **mag=** (double m)

Description: Sets the magnification

m: The new magnification

Python specific notes:
The object exposes a writable attribute 'mag'. This is the setter.

mirror=

Signature: void **mirror=** (bool m)

Description: Sets the mirror flag

m: The new mirror flag

"mirroring" describes a reflection at the x-axis which is included in the transformation prior to rotation.

Python specific notes:
The object exposes a writable attribute 'mirror'. This is the setter.

new

(1) Signature: *[static]* new ICplxTrans ptr **new** (const DCplxTrans trans, double dbu = 1)

Description: Creates an integer coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the input space and output space from floating-point units to integer units and vice versa. Formally, the ICplxTrans transformation is initialized with 'to_dbu * trans * from_dbu' where 'from_dbu' is the transformation into micrometer space, or more

precisely 'CplxTrans(mag=dbu)' and 'to_dbu' is the transformation into DBU space, or more precisely 'VCplxTrans(mag=1/dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new ICplxTrans ptr **new** (const CplxTrans trans, double dbu = 1)

Description: Creates an integer coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the output space from floating-point units to integer units. Formally, the CplxTrans transformation is initialized with 'to_dbu * trans' where 'to_dbu' is the transformation into DBU space, or more precisely 'VCplxTrans(mag=1/dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new ICplxTrans ptr **new** (const VCplxTrans trans, double dbu = 1)

Description: Creates an integer coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the input space from floating-point units to integer units. Formally, the CplxTrans transformation is initialized with 'trans * from_dbu' where 'from_dbu' is the transformation into micrometer space, or more precisely 'CplxTrans(mag=dbu)'.

This constructor has been introduced in version 0.25. The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new ICplxTrans ptr **new**

Description: Creates a unit transformation

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new ICplxTrans ptr **new** (const ICplxTrans c, double mag = 1, const Vector u = 0,0)

Description: Creates a transformation from another transformation plus a magnification and displacement

c:	The original transformation
u:	The Additional displacement

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new ICplxTrans ptr **new** (const ICplxTrans c, double mag = 1, int x = 0, int y = 0)

Description: Creates a transformation from another transformation plus a magnification and displacement

c:	The original transformation
x:	The Additional displacement (x)
y:	The Additional displacement (y)

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new ICplxTrans ptr **new** (int x, int y)

Description: Creates a transformation from a x and y displacement

x:	The x displacement
y:	The y displacement

This constructor will create a transformation with the specified displacement but no rotation.

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new ICplxTrans ptr **new** (const Trans t, double mag = 1)

Description: Creates a transformation from a simple transformation and a magnification

Creates a magnifying transformation from a simple transformation and a magnification.

Python specific notes:
This method is the default initializer of the object.

(9) Signature: *[static]* new ICplxTrans ptr **new** (const Vector u)

Description: Creates a transformation from a displacement

Creates a transformation with a displacement only.

This method has been added in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(10) Signature: *[static]* new ICplxTrans ptr **new** (double mag = 1, double rot = 0, bool mirrx = false, const Vector u = 0,0)

Description: Creates a transformation using magnification, angle, mirror flag and displacement

mag:	The magnification
rot:	The rotation angle in units of degree
mirrx:	True, if mirrored at x axis
u:	The displacement

The sequence of operations is: magnification, mirroring at x axis, rotation, application of displacement.

Python specific notes:
This method is the default initializer of the object.

(11) Signature: *[static]* new ICplxTrans ptr **new** (double mag = 1, double rot = 0, bool mirrx = false, int x = 0, int y = 0)

Description: Creates a transformation using magnification, angle, mirror flag and displacement

mag:	The magnification
rot:	The rotation angle in units of degree
mirrx:	True, if mirrored at x axis
x:	The x displacement
y:	The y displacement

The sequence of operations is: magnification, mirroring at x axis, rotation, application of displacement.

Python specific notes:
This method is the default initializer of the object.

rot

Signature: *[const]* int **rot**

Description: Returns the respective simple transformation equivalent rotation code if possible

If this transformation is orthogonal (is_ortho () == true), then this method will return the corresponding fixpoint transformation, not taking into account magnification and displacement. If the transformation is not orthogonal, the result reflects the quadrant the rotation goes into.

s_trans

Signature: *[const]* Trans **s_trans**

Description: Extracts the simple transformation part

The simple transformation part does not reflect magnification or arbitrary angles. Rotation angles are rounded down to multiples of 90 degree. Magnification is fixed to 1.0.

to_itrans

Signature: *[const]* DCplxTrans **to_itrans** (double dbu = 1)

Description: Converts the transformation to another transformation with floating-point input and output coordinates

Use of this method is deprecated

The database unit can be specified to translate the integer coordinate displacement in database units to a floating-point displacement in micron units. The displacement's' coordinates will be multiplied with the database unit.

This method is redundant with the conversion constructors and is ill-named. Instead of 'to_itrans' use the conversion constructor:

```
dtrans = RBA::DCplxTrans::new(itrans, dbu)
```

This method has been introduced in version 0.25 and was deprecated in version 0.29.

to_s

Signature: *[const]* string **to_s** (bool lazy = false, double dbu = 0)

Description: String conversion

If 'lazy' is true, some parts are omitted when not required. If a DBU is given, the output units will be micrometers.

The lazy and DBU arguments have been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

to_trans

Signature: *[const]* VCplxTrans **to_trans** (double dbu = 1)

Description: Converts the transformation to another transformation with floating-point input coordinates

Use of this method is deprecated

This method is redundant with the conversion constructors and is ill-named. Instead of 'to_trans' use the conversion constructor:

```
vtrans = RBA::VCplxTrans::new(itrans, dbu)
```

This method has been introduced in version 0.25 and was deprecated in version 0.29.

to_vtrans

Signature: *[const]* CplxTrans **to_vtrans** (double dbu = 1)

Description: Converts the transformation to another transformation with floating-point output coordinates

Use of this method is deprecated

The database unit can be specified to translate the integer coordinate displacement in database units to a floating-point displacement in micron units. The displacement's' coordinates will be multiplied with the database unit.

This method is redundant with the conversion constructors and is ill-named. Instead of 'to_vtrans' use the conversion constructor:

```
trans = RBA::CplxTrans::new(itrans, dbu)
```

This method has been introduced in version 0.25 and was deprecated in version 0.29.

trans

(1) Signature: *[const]* Point **trans** (const Point p)

Description: Transforms a point

p:	The point to transform
Returns:	The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* Vector **trans** (const Vector p)

Description: Transforms a vector

v:	The vector to transform
Returns:	The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* Box **trans** (const Box box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* Edge **trans** (const Edge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* Polygon **trans** (const Polygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform
Returns:	The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* Path **trans** (const Path path)

Description: Transforms a path

path:	The path to transform
Returns:	The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* Text **trans** (const Text text)

Description: Transforms a text

text:	The text to transform
Returns:	The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

4.110. API reference - Class VCplxTrans

Notation used in Ruby API documentation

Module: db

Description: A complex transformation

A complex transformation provides magnification, mirroring at the x-axis, rotation by an arbitrary angle and a displacement. This is also the order, the operations are applied. This version can transform floating point coordinate objects into integer coordinate objects, which may involve rounding and can be inexact.

Complex transformations are extensions of the simple transformation classes (Trans in that case) and behave similar.

Transformations can be used to transform points or other objects. Transformations can be combined with the '*' operator to form the transformation which is equivalent to applying the second and then the first. Here is some code:

```
# Create a transformation that applies a magnification of 1.5, a rotation by 90 degree
# and displacement of 10 in x and 20 units in y direction:
t = RBA::VCplxTrans::new(1.5, 90, false, 10, 20)
t.to_s            # r90 *1.5 10,20
# compute the inverse:
t.inverted.to_s   # r270 *0.666666667 -13,7
# Combine with another displacement (applied after that):
(RBA::VCplxTrans::new(5, 5) * t).to_s     # r90 *1.5 15,25
# Transform a point:
t.trans(RBA::DPoint::new(100, 200)).to_s  # -290,170
```

The VCplxTrans type is the inverse transformation of the CplxTrans transformation and vice versa.Transformations of VCplxTrans type can be concatenated (operator *) with either itself or with transformations of compatible input or output type. This means, the operator VCplxTrans * CplxTrans is allowed (output types of CplxTrans and input of VCplxTrans are identical) while VCplxTrans * ICplxTrans is not.

This class has been introduced in version 0.25.

See The Database API for more details about the database objects.

Public constructors

new VCplxTrans ptr	**new**	(const DCplxTrans trans, double dbu = 1)	Creates a floating-point to integer coordinate transformation from another coordinate flavour
new VCplxTrans ptr	**new**	(const CplxTrans trans, double dbu = 1)	Creates a floating-point to integer coordinate transformation from another coordinate flavour
new VCplxTrans ptr	**new**	(const ICplxTrans trans, double dbu = 1)	Creates a floating-point to integer coordinate transformation from another coordinate flavour
new VCplxTrans ptr	**new**		Creates a unit transformation
new VCplxTrans ptr	**new**	(const VCplxTrans c, double mag = 1, const Vector u = 0,0)	Creates a transformation from another transformation plus a magnification and displacement
new VCplxTrans ptr	**new**	(const VCplxTrans c, double mag = 1, int x = 0, int y = 0)	Creates a transformation from another transformation plus a magnification and displacement
new VCplxTrans ptr	**new**	(int x, int y)	Creates a transformation from a x and y displacement

new VCplxTrans ptr	new	(const DTrans t, double mag = 1)	Creates a transformation from a simple transformation and a magnification
new VCplxTrans ptr	new	(const Vector u)	Creates a transformation from a displacement
new VCplxTrans ptr	new	(double mag = 1, double rot = 0, bool mirrx = false, const Vector u = 0,0)	Creates a transformation using magnification, angle, mirror flag and displacement
new VCplxTrans ptr	new	(double mag = 1, double rot = 0, bool mirrx = false, int x = 0, int y = 0)	Creates a transformation using magnification, angle, mirror flag and displacement

Public methods

[const]	bool	!=	(const VCplxTrans other)	Tests for inequality
[const]	VCplxTrans	*	(const DCplxTrans t)	Multiplication (concatenation) of transformations
[const]	ICplxTrans	*	(const CplxTrans t)	Multiplication (concatenation) of transformations
[const]	unsigned int	*	(double d)	Transforms a distance
[const]	Point	*	(const DPoint p)	Transforms a point
[const]	Vector	*	(const DVector p)	Transforms a vector
[const]	Box	*	(const DBox box)	Transforms a box
[const]	Edge	*	(const DEdge edge)	Transforms an edge
[const]	Polygon	*	(const DPolygon polygon)	Transforms a polygon
[const]	Path	*	(const DPath path)	Transforms a path
[const]	Text	*	(const DText text)	Transforms a text
[const]	VCplxTrans	*	(const VCplxTrans t)	Returns the concatenated transformation
[const]	bool	<	(const VCplxTrans other)	Provides a 'less' criterion for sorting
[const]	bool	==	(const VCplxTrans other)	Tests for equality
	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object

[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	double	angle		Gets the angle
	void	angle=	(double a)	Sets the angle
	void	assign	(const VCplxTrans other)	Assigns another object to self
[const]	unsigned int	ctrans	(double d)	Transforms a distance
[const]	Vector	disp		Gets the displacement
	void	disp=	(const Vector u)	Sets the displacement
[const]	new VCplxTrans ptr	dup		Creates a copy of self
[const]	unsigned long	hash		Computes a hash value
	VCplxTrans	invert		Inverts the transformation (in place)
[const]	CplxTrans	inverted		Returns the inverted transformation
[const]	bool	is_complex?		Returns true if the transformation is a complex one
[const]	bool	is_mag?		Tests, if the transformation is a magnifying one
[const]	bool	is_mirror?		Gets the mirror flag
[const]	bool	is_ortho?		Tests, if the transformation is an orthogonal transformation
[const]	bool	is_unity?		Tests, whether this is a unit transformation
[const]	double	mag		Gets the magnification
	void	mag=	(double m)	Sets the magnification
	void	mirror=	(bool m)	Sets the mirror flag
[const]	int	rot		Returns the respective simple transformation equivalent rotation code if possible
[const]	DTrans	s_trans		Extracts the simple transformation part

[const]	string	**to_s**	(bool lazy = false, double dbu = 0)	String conversion
[const]	Point	**trans**	(const DPoint p)	Transforms a point
[const]	Vector	**trans**	(const DVector p)	Transforms a vector
[const]	Box	**trans**	(const DBox box)	Transforms a box
[const]	Edge	**trans**	(const DEdge edge)	Transforms an edge
[const]	Polygon	**trans**	(const DPolygon polygon)	Transforms a polygon
[const]	Path	**trans**	(const DPath path)	Transforms a path
[const]	Text	**trans**	(const DText text)	Transforms a text

Public static methods and constants

VCplxTrans	**M0**		A constant giving "mirrored at the x-axis" transformation
VCplxTrans	**M135**		A constant giving "mirrored at the 135 degree axis" transformation
VCplxTrans	**M45**		A constant giving "mirrored at the 45 degree axis" transformation
VCplxTrans	**M90**		A constant giving "mirrored at the y (90 degree) axis" transformation
VCplxTrans	**R0**		A constant giving "unrotated" (unit) transformation
VCplxTrans	**R180**		A constant giving "rotated by 180 degree counterclockwise" transformation
VCplxTrans	**R270**		A constant giving "rotated by 270 degree counterclockwise" transformation
VCplxTrans	**R90**		A constant giving "rotated by 90 degree counterclockwise" transformation
new VCplxTrans ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead

[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	DCplxTrans	**to_itrans**	(double dbu = 1)	Use of this method is deprecated
[const]	ICplxTrans	**to_trans**	(double arg1)	Use of this method is deprecated
[const]	CplxTrans	**to_vtrans**	(double dbu = 1)	Use of this method is deprecated

Detailed description

!=

Signature: *[const]* bool **!=** (const VCplxTrans other)

Description: Tests for inequality

*****

(1) Signature: *[const]* VCplxTrans ***** (const DCplxTrans t)

Description: Multiplication (concatenation) of transformations

t: The transformation to apply before

Returns: The modified transformation

The * operator returns self*t ("t is applied before this transformation").

(2) Signature: *[const]* ICplxTrans ***** (const CplxTrans t)

Description: Multiplication (concatenation) of transformations

t: The transformation to apply before

Returns: The modified transformation

The * operator returns self*t ("t is applied before this transformation").

(3) Signature: *[const]* unsigned int ***** (double d)

Description: Transforms a distance

d: The distance to transform

Returns: The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* Point ***** (const DPoint p)

Description: Transforms a point

p: The point to transform

Returns: The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:

This method also implements '__rmul__'.

(5) Signature: *[const]* Vector ***** (const DVector p)

Description: Transforms a vector

v:	The vector to transform
Returns:	The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* Box ***** (const DBox box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* Edge ***** (const DEdge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(8) Signature: *[const]* Polygon ***** (const DPolygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform
Returns:	The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(9) Signature: *[const]* Path ***** (const DPath path)

Description: Transforms a path

path:	The path to transform
Returns:	The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(10) Signature: *[const]* Text ***** (const DText text)

Description: Transforms a text

text:	The text to transform
Returns:	The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(11) Signature: *[const]* VCplxTrans ***** (const VCplxTrans t)

Description: Returns the concatenated transformation

t:	The transformation to apply before
Returns:	The modified transformation

The * operator returns self*t ("t is applied before this transformation").

<

Signature: *[const]* bool **<** (const VCplxTrans other)

Description: Provides a 'less' criterion for sorting

This method is provided to implement a sorting order. The definition of 'less' is opaque and might change in future versions.

==

Signature: *[const]* bool **==** (const VCplxTrans other)

Description: Tests for equality

M0

Signature: *[static]* VCplxTrans **M0**

Description: A constant giving "mirrored at the x-axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M0'. This is the getter.

M135

Signature: *[static]* VCplxTrans **M135**

Description: A constant giving "mirrored at the 135 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M135'. This is the getter.

M45

Signature: *[static]* VCplxTrans **M45**

Description: A constant giving "mirrored at the 45 degree axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M45'. This is the getter.

M90

Signature: *[static]* VCplxTrans **M90**

Description: A constant giving "mirrored at the y (90 degree) axis" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'M90'. This is the getter.

R0

Signature: *[static]* VCplxTrans **R0**

Description: A constant giving "unrotated" (unit) transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R0'. This is the getter.

R180

Signature: *[static]* VCplxTrans **R180**

Description: A constant giving "rotated by 180 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R180'. This is the getter.

R270

Signature: *[static]* VCplxTrans **R270**

Description: A constant giving "rotated by 270 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R270'. This is the getter.

R90

Signature: *[static]* VCplxTrans **R90**

Description: A constant giving "rotated by 90 degree counterclockwise" transformation

The previous integer constant has been turned into a transformation in version 0.25.

Python specific notes:
The object exposes a readable attribute 'R90'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

angle

Signature: *[const]* double **angle**

Description: Gets the angle

Returns: The rotation angle this transformation provides in degree units (0..360 deg).

Note that the simple transformation returns the angle in units of 90 degree. Hence for a simple trans (i.e. Trans), a rotation angle of 180 degree delivers a value of 2 for the angle attribute. The complex transformation, supporting any rotation angle returns the angle in degree.

Python specific notes:
The object exposes a readable attribute 'angle'. This is the getter.

angle=

Signature: void **angle=** (double a)

Description: Sets the angle

a: The new angle

See angle for a description of that attribute.

Python specific notes:
The object exposes a writable attribute 'angle'. This is the setter.

assign

Signature: void **assign** (const VCplxTrans other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

ctrans

Signature: *[const]* unsigned int **ctrans** (double d)

Description: Transforms a distance

d: The distance to transform

Returns: The transformed distance

The "ctrans" method transforms the given distance. e = t(d). For the simple transformations, there is no magnification and no modification of the distance therefore.

The product '*' has been added as a synonym in version 0.28.

Python specific notes:
This method also implements '__rmul__'.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disp

Signature: *[const]* Vector **disp**

Description: Gets the displacement

Python specific notes:
The object exposes a readable attribute 'disp'. This is the getter.

disp=

Signature: void **disp=** (const Vector u)

Description: Sets the displacement

u: The new displacement

Python specific notes:
The object exposes a writable attribute 'disp'. This is the setter.

dup

Signature: *[const]* new VCplxTrans ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_s

Signature: *[static]* new VCplxTrans ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

This method has been added in version 0.23.

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given transformation. This method enables transformations as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

invert

Signature: VCplxTrans **invert**

Description: Inverts the transformation (in place)

Returns: The inverted transformation

Inverts the transformation and replaces this transformation by its inverted one.

inverted

Signature: *[const]* CplxTrans **inverted**

Description: Returns the inverted transformation

Returns: The inverted transformation

Returns the inverted transformation. This method does not modify the transformation.

is_complex?

Signature: *[const]* bool **is_complex?**

Description: Returns true if the transformation is a complex one

If this predicate is false, the transformation can safely be converted to a simple transformation. Otherwise, this conversion will be lossy. The predicate value is equivalent to 'is_mag || !is_ortho'.

This method has been introduced in version 0.27.5.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_mag?

Signature: *[const]* bool **is_mag?**

Description: Tests, if the transformation is a magnifying one

This is the recommended test for checking if the transformation represents a magnification.

is_mirror?

Signature: *[const]* bool **is_mirror?**

Description: Gets the mirror flag

If this property is true, the transformation is composed of a mirroring at the x-axis followed by a rotation by the angle given by the angle property.

Python specific notes:
The object exposes a readable attribute 'mirror'. This is the getter.

is_ortho?

Signature: *[const]* bool **is_ortho?**

Description: Tests, if the transformation is an orthogonal transformation

If the rotation is by a multiple of 90 degree, this method will return true.

is_unity?

Signature: *[const]* bool **is_unity?**

Description: Tests, whether this is a unit transformation

mag

Signature: *[const]* double **mag**

Description: Gets the magnification

Python specific notes:
The object exposes a readable attribute 'mag'. This is the getter.

mag=

Signature: void **mag=** (double m)

Description: Sets the magnification

m: The new magnification

Python specific notes:
The object exposes a writable attribute 'mag'. This is the setter.

mirror=

Signature: void **mirror=** (bool m)

Description: Sets the mirror flag

m: The new mirror flag

"mirroring" describes a reflection at the x-axis which is included in the transformation prior to rotation.

Python specific notes:
The object exposes a writable attribute 'mirror'. This is the setter.

new

(1) Signature: *[static]* new VCplxTrans ptr **new** (const DCplxTrans trans, double dbu = 1)

Description: Creates a floating-point to integer coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the output space from floating-point units to integer units. Formally, the VCplxTrans transformation is initialized with 'to_dbu * trans' where 'to_dbu' is the transformation into DBU space, or more precisely 'VCplxTrans(mag=1/dbu)'.

The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new VCplxTrans ptr **new** (const CplxTrans trans, double dbu = 1)

Description: Creates a floating-point to integer coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the input and output space from floating-point units to integer units and vice versa. Formally, the VCplxTrans transformation is initialized with 'to_dbu * trans * to_dbu' where 'to_dbu' is the transformation into DBU space, or more precisely 'VCplxTrans(mag=1/dbu)'.

The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new VCplxTrans ptr **new** (const ICplxTrans trans, double dbu = 1)

Description: Creates a floating-point to integer coordinate transformation from another coordinate flavour

The 'dbu' argument is used to transform the input and output space from floating-point units to integer units and vice versa. Formally, the VCplxTrans transformation is initialized with 'trans * to_dbu' where 'to_dbu' is the transformation into DBU space, or more precisely 'VCplxTrans(mag=1/dbu)'.

The 'dbu' argument has been added in version 0.29.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new VCplxTrans ptr **new**

Description: Creates a unit transformation

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new VCplxTrans ptr **new** (const VCplxTrans c, double mag = 1, const Vector u = 0,0)

Description: Creates a transformation from another transformation plus a magnification and displacement

c:	The original transformation
u:	The Additional displacement

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new VCplxTrans ptr **new** (const VCplxTrans c, double mag = 1, int x = 0, int y = 0)

Description: Creates a transformation from another transformation plus a magnification and displacement

c:	The original transformation
x:	The Additional displacement (x)
y:	The Additional displacement (y)

Creates a new transformation from a existing transformation. This constructor is provided for creating duplicates and backward compatibility since the constants are transformations now. It will copy the original transformation and add the given displacement.

This variant has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new VCplxTrans ptr **new** (int x, int y)

Description: Creates a transformation from a x and y displacement

x:	The x displacement
y:	The y displacement

This constructor will create a transformation with the specified displacement but no rotation.

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new VCplxTrans ptr **new** (const DTrans t, double mag = 1)

Description: Creates a transformation from a simple transformation and a magnification

Creates a magnifying transformation from a simple transformation and a magnification.

Python specific notes:
This method is the default initializer of the object.

(9) Signature: *[static]* new VCplxTrans ptr **new** (const Vector u)

Description: Creates a transformation from a displacement

Creates a transformation with a displacement only.

This method has been added in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(10) Signature: *[static]* new VCplxTrans ptr **new** (double mag = 1, double rot = 0, bool mirrx = false, const Vector u = 0,0)

Description: Creates a transformation using magnification, angle, mirror flag and displacement

mag:	The magnification
rot:	The rotation angle in units of degree
mirrx:	True, if mirrored at x axis
u:	The displacement

The sequence of operations is: magnification, mirroring at x axis, rotation, application of displacement.

Python specific notes:
This method is the default initializer of the object.

(11) Signature: *[static]* new VCplxTrans ptr **new** (double mag = 1, double rot = 0, bool mirrx = false, int x = 0, int y = 0)

Description: Creates a transformation using magnification, angle, mirror flag and displacement

mag:	The magnification
rot:	The rotation angle in units of degree
mirrx:	True, if mirrored at x axis
x:	The x displacement
y:	The y displacement

The sequence of operations is: magnification, mirroring at x axis, rotation, application of displacement.

Python specific notes:
This method is the default initializer of the object.

rot

Signature: *[const]* int **rot**

Description: Returns the respective simple transformation equivalent rotation code if possible

If this transformation is orthogonal (is_ortho () == true), then this method will return the corresponding fixpoint transformation, not taking into account magnification and displacement. If the transformation is not orthogonal, the result reflects the quadrant the rotation goes into.

s_trans

Signature: *[const]* DTrans **s_trans**

Description: Extracts the simple transformation part

The simple transformation part does not reflect magnification or arbitrary angles. Rotation angles are rounded down to multiples of 90 degree. Magnification is fixed to 1.0.

to_itrans

Signature: *[const]* DCplxTrans **to_itrans** (double dbu = 1)

Description: Converts the transformation to another transformation with floating-point output coordinates

Use of this method is deprecated

The database unit can be specified to translate the integer coordinate displacement in database units to a floating-point displacement in micron units. The displacement's' coordinates will be multiplied with the database unit.

This method is redundant with the conversion constructors and is ill-named. Instead of 'to_itrans' use the conversion constructor:

```
dtrans = RBA::DCplxTrans::new(vtrans, dbu)
```

This method has been deprecated in version 0.29.

to_s

Signature: *[const]* string **to_s** (bool lazy = false, double dbu = 0)

Description: String conversion

If 'lazy' is true, some parts are omitted when not required. If a DBU is given, the output units will be micrometers.

The lazy and DBU arguments have been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

to_trans

Signature: *[const]* ICplxTrans **to_trans** (double arg1)

Description: Converts the transformation to another transformation with integer input coordinates

Use of this method is deprecated

This method is redundant with the conversion constructors and is ill-named. Instead of 'to_trans' use the conversion constructor:

```
itrans = RBA::ICplxTrans::new(vtrans, dbu)
```

This method has been deprecated in version 0.29.

to_vtrans

Signature: *[const]* CplxTrans **to_vtrans** (double dbu = 1)

Description: Converts the transformation to another transformation with integer input and floating-point output coordinates

Use of this method is deprecated

The database unit can be specified to translate the integer coordinate displacement in database units to an floating-point displacement in micron units. The displacement's' coordinates will be multiplied with the database unit.

This method is redundant with the conversion constructors and is ill-named. Instead of 'to_vtrans' use the conversion constructor:

```
trans = RBA::CplxTrans::new(vtrans, dbu)
```

This method has been deprecated in version 0.29.

trans

(1) Signature: *[const]* Point **trans** (const DPoint p)

Description: Transforms a point

p:	The point to transform
Returns:	The transformed point

The "trans" method or the * operator transforms the given point. q = t(p)

The * operator has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* Vector **trans** (const DVector p)

Description: Transforms a vector

v:	The vector to transform
Returns:	The transformed vector

The "trans" method or the * operator transforms the given vector. w = t(v)

Vector transformation has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(3) Signature: *[const]* Box **trans** (const DBox box)

Description: Transforms a box

box:	The box to transform
Returns:	The transformed box

't*box' or 't.trans(box)' is equivalent to box.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(4) Signature: *[const]* Edge **trans** (const DEdge edge)

Description: Transforms an edge

edge:	The edge to transform
Returns:	The transformed edge

't*edge' or 't.trans(edge)' is equivalent to edge.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(5) Signature: *[const]* Polygon **trans** (const DPolygon polygon)

Description: Transforms a polygon

polygon:	The polygon to transform

Returns: The transformed polygon

't*polygon' or 't.trans(polygon)' is equivalent to polygon.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(6) Signature: *[const]* Path **trans** (const DPath path)

Description: Transforms a path

path: The path to transform

Returns: The transformed path

't*path' or 't.trans(path)' is equivalent to path.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

(7) Signature: *[const]* Text **trans** (const DText text)

Description: Transforms a text

text: The text to transform

Returns: The transformed text

't*text' or 't.trans(text)' is equivalent to text.transformed(t).

This convenience method has been introduced in version 0.25.

Python specific notes:
This method also implements '__rmul__'.

4.111. API reference - Class Utils

Notation used in Ruby API documentation

Module: db

Description: This namespace provides a collection of utility functions

This class has been introduced in version 0.27.

Public constructors

new Utils ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Utils other)	Assigns another object to self
[const]	new Utils ptr	**dup**		Creates a copy of self

Public static methods and constants

	DPoint[]	**spline_interpolation**	(DPoint[] control_points, double[] weights, int degree, double[] knots, double relative_accuracy, double absolute_accuracy)	This function computes the Spline curve for a given set of control points (point, weight), degree and knots.
	Point[]	**spline_interpolation**	(Point[] control_points, double[] weights, int degree, double[] knots, double relative_accuracy, double absolute_accuracy)	This function computes the Spline curve for a given set of control points (point, weight), degree and knots.
	DPoint[]	**spline_interpolation**	(DPoint[] control_points, int degree, double[] knots, double relative_accuracy,	This function computes the Spline curve for a given set of control points (point, weight), degree and knots.

		double absolute_accuracy)	
Point[]	spline_interpolation	(Point[] control_points, int degree, double[] knots, double relative_accuracy, double absolute_accuracy)	This function computes the Spline curve for a given set of control points (point, weight), degree and knots.

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void _**manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

	Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
assign	**Signature**: void **assign** (const Utils other) **Description**: Assigns another object to self
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
dup	**Signature**: *[const]* new Utils ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new Utils ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

spline_interpolation

(1) Signature: *[static]* DPoint[] **spline_interpolation** (DPoint[] control_points, double[] weights, int degree, double[] knots, double relative_accuracy, double absolute_accuracy)

Description: This function computes the Spline curve for a given set of control points (point, weight), degree and knots.

The knot vector needs to be padded and its size must fulfill the condition:

```
knots.size == control_points.size + degree + 1
```

The accuracy parameters allow tuning the resolution of the curve to target a specific approximation quality. "relative_accuracy" gives the accuracy relative to the local curvature radius, "absolute" accuracy gives the absolute accuracy. "accuracy" is the allowed deviation of polygon approximation from the ideal curve. The computed curve should meet at least one of the accuracy criteria. Setting both limits to a very small value will result in long run times and a large number of points returned.

This function supports both rational splines (NURBS) and non-rational splines. The latter use weights of 1.0 for each point.

The return value is a list of points forming a path which approximates the spline curve.

(2) Signature: *[static]* Point[] **spline_interpolation** (Point[] control_points, double[] weights, int degree, double[] knots, double relative_accuracy, double absolute_accuracy)

Description: This function computes the Spline curve for a given set of control points (point, weight), degree and knots.

This is the version for integer-coordinate points.

(3) Signature: *[static]* DPoint[] **spline_interpolation** (DPoint[] control_points, int degree, double[] knots, double relative_accuracy, double absolute_accuracy)

Description: This function computes the Spline curve for a given set of control points (point, weight), degree and knots.

This is the version for non-rational splines. It lacks the weight vector.

(4) Signature: *[static]* Point[] **spline_interpolation** (Point[] control_points, int degree, double[] knots, double relative_accuracy, double absolute_accuracy)

Description: This function computes the Spline curve for a given set of control points (point, weight), degree and knots.

This is the version for integer-coordinate points for non-rational splines.

4.112. API reference - Class DVector

Notation used in Ruby API documentation

Module: db

Description: A vector class with double (floating-point) coordinates

A vector is a distance in cartesian, 2 dimensional space. A vector is given by two coordinates (x and y) and represents the distance between two points. Being the distance, transformations act differently on vectors: the displacement is not applied. Vectors are not geometrical objects by itself. But they are frequently used in the database API for various purposes. Other than the integer variant (Vector), points with floating-point coordinates can represent fractions of a database unit or vectors in physical (micron) units.

This class has been introduced in version 0.25.

See The Database API for more details about the database objects.

Public constructors

new DVector ptr	**new**	(const Vector vector)	Creates a floating-point coordinate vector from an integer coordinate vector
new DVector ptr	**new**		Default constructor: creates a null vector with coordinates (0,0)
new DVector ptr	**new**	(const DPoint p)	Default constructor: creates a vector from a point
new DVector ptr	**new**	(double x, double y)	Constructor for a vector from two coordinate values

Public methods

[const]	bool	**!=**	(const DVector v)	Inequality test operator
[const]	DVector	*****	(double f)	Scaling by some factor
[const]	double	*****	(const DVector v)	Computes the scalar product between self and the given vector
	DVector	***=**	(double f)	Scaling by some factor
[const]	DVector	**+**	(const DVector v)	Adds two vectors
[const]	DPoint	**+**	(const DPoint p)	Adds a vector and a point
[const]	DVector	**-**	(const DVector v)	Subtract two vectors
[const]	DVector	**-@**		Compute the negative of a vector
[const]	DVector	***/***	(double d)	Division by some divisor
	DVector	***/=**	(double d)	Division by some divisor
[const]	bool	**<**	(const DVector v)	"less" comparison operator
[const]	bool	**==**	(const DVector v)	Equality test operator
	void	**_create**		Ensures the C++ object is created

	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	double	abs		Returns the length of the vector
	void	assign	(const DVector other)	Assigns another object to self
[const]	new DVector ptr	dup		Creates a copy of self
[const]	unsigned long	hash		Computes a hash value
[const]	double	length		Returns the length of the vector
[const]	double	sprod	(const DVector v)	Computes the scalar product between self and the given vector
[const]	int	sprod_sign	(const DVector v)	Computes the scalar product between self and the given vector and returns a value indicating the sign of the product
[const]	double	sq_abs		The square length of the vector
[const]	double	sq_length		The square length of the vector
[const]	Vector	to_itype	(double dbu = 1)	Converts the point to an integer coordinate point
[const]	DPoint	to_p		Turns the vector into a point
[const]	string	to_s	(double dbu = 0)	String conversion
[const]	double	vprod	(const DVector v)	Computes the vector product between self and the given vector
[const]	int	vprod_sign	(const DVector v)	Computes the vector product between self and the given vector and returns a value indicating the sign of the product
[const]	double	x		Accessor to the x coordinate
	void	x=	(double coord)	Write accessor to the x coordinate
[const]	double	y		Accessor to the y coordinate
	void	y=	(double coord)	Write accessor to the y coordinate

Public static methods and constants

new DVector ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const DVector v)

Description: Inequality test operator

*****

(1) Signature: *[const]* DVector ***** (double f)

Description: Scaling by some factor

Returns the scaled object. All coordinates are multiplied with the given factor and if necessary rounded.

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* double ***** (const DVector v)

Description: Computes the scalar product between self and the given vector

The scalar product of a and b is defined as: vp = ax*bx+ay*by.

Python specific notes:
This method also implements '__rmul__'.

***=**

Signature: DVector ***=** (double f)

Description: Scaling by some factor

Scales object in place. All coordinates are multiplied with the given factor and if necessary rounded.

+

(1) Signature: *[const]* DVector **+** (const DVector v)

Description: Adds two vectors

Adds vector v to self by adding the coordinates.

(2) Signature: *[const]* DPoint **+** (const DPoint p)

Description: Adds a vector and a point

Returns the point p shifted by the vector.

-	**Signature**: *[const]* DVector **-** (const DVector v) **Description**: Subtract two vectors Subtract vector v from self by subtracting the coordinates.
-@	**Signature**: *[const]* DVector **-@** **Description**: Compute the negative of a vector Returns a new vector with -x,-y.
/	**Signature**: *[const]* DVector **/** (double d) **Description**: Division by some divisor Returns the scaled object. All coordinates are divided with the given divisor and if necessary rounded.
/=	**Signature**: DVector **/=** (double d) **Description**: Division by some divisor Divides the object in place. All coordinates are divided with the given divisor and if necessary rounded.
<	**Signature**: *[const]* bool **<** (const DVector v) **Description**: "less" comparison operator This operator is provided to establish a sorting order
==	**Signature**: *[const]* bool **==** (const DVector v) **Description**: Equality test operator
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

abs

Signature: *[const]* double **abs**

Description: Returns the length of the vector

'abs' is an alias provided for compatibility with the former point type.

assign

Signature: void **assign** (const DVector other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new DVector ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_s

Signature: *[static]* new DVector ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given vector. This method enables vectors as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

length

Signature: *[const]* double **length**

Description: Returns the length of the vector

'abs' is an alias provided for compatibility with the former point type.

new

(1) Signature: *[static]* new DVector ptr **new** (const Vector vector)

Description: Creates a floating-point coordinate vector from an integer coordinate vector

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new DVector ptr **new**

Description: Default constructor: creates a null vector with coordinates (0,0)

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new DVector ptr **new** (const DPoint p)

Description: Default constructor: creates a vector from a point

This constructor is equivalent to computing p-point(0,0). This method has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new DVector ptr **new** (double x, double y)

Description: Constructor for a vector from two coordinate values

Python specific notes:
This method is the default initializer of the object.

sprod

Signature: *[const]* double **sprod** (const DVector v)

Description: Computes the scalar product between self and the given vector

The scalar product of a and b is defined as: vp = ax*bx+ay*by.

sprod_sign

Signature: *[const]* int **sprod_sign** (const DVector v)

Description: Computes the scalar product between self and the given vector and returns a value indicating the sign of the product

Returns: 1 if the scalar product is positive, 0 if it is zero and -1 if it is negative.

sq_abs

Signature: *[const]* double **sq_abs**

Description: The square length of the vector

'sq_abs' is an alias provided for compatibility with the former point type.

sq_length

Signature: *[const]* double **sq_length**

Description: The square length of the vector

'sq_abs' is an alias provided for compatibility with the former point type.

to_itype

Signature: *[const]* Vector **to_itype** (double dbu = 1)

Description: Converts the point to an integer coordinate point

The database unit can be specified to translate the floating-point coordinate vector in micron units to an integer-coordinate vector in database units. The vector's' coordinates will be divided by the database unit.

to_p

Signature: *[const]* DPoint **to_p**

Description: Turns the vector into a point

This method returns the point resulting from adding the vector to (0,0). This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: String conversion

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

vprod

Signature: *[const]* double **vprod** (const DVector v)

Description: Computes the vector product between self and the given vector

The vector product of a and b is defined as: vp = ax*by-ay*bx.

vprod_sign

Signature: *[const]* int **vprod_sign** (const DVector v)

Description: Computes the vector product between self and the given vector and returns a value indicating the sign of the product

Returns: 1 if the vector product is positive, 0 if it is zero and -1 if it is negative.

x

Signature: *[const]* double **x**

Description: Accessor to the x coordinate

Python specific notes:
The object exposes a readable attribute 'x'. This is the getter.

x=

Signature: void **x=** (double coord)

Description: Write accessor to the x coordinate

Python specific notes:
The object exposes a writable attribute 'x'. This is the setter.

y

Signature: *[const]* double **y**

Description: Accessor to the y coordinate

Python specific notes:
The object exposes a readable attribute 'y'. This is the getter.

y=

Signature: void **y=** (double coord)

Description: Write accessor to the y coordinate

Python specific notes:
The object exposes a writable attribute 'y'. This is the setter.

4.113. API reference - Class Vector

Notation used in Ruby API documentation

Module: db

Description: A integer vector class

A vector is a distance in cartesian, 2 dimensional space. A vector is given by two coordinates (x and y) and represents the distance between two points. Being the distance, transformations act differently on vectors: the displacement is not applied. Vectors are not geometrical objects by itself. But they are frequently used in the database API for various purposes.

This class has been introduced in version 0.25.

See The Database API for more details about the database objects.

Public constructors

new Vector ptr	**new**	(const DVector dvector)	Creates an integer coordinate vector from a floating-point coordinate vector
new Vector ptr	**new**		Default constructor: creates a null vector with coordinates (0,0)
new Vector ptr	**new**	(const Point p)	Default constructor: creates a vector from a point
new Vector ptr	**new**	(int x, int y)	Constructor for a vector from two coordinate values

Public methods

[const]	bool	**!=**	(const Vector v)	Inequality test operator
[const]	Vector	*****	(double f)	Scaling by some factor
[const]	long	*****	(const Vector v)	Computes the scalar product between self and the given vector
	Vector	***=**	(double f)	Scaling by some factor
[const]	Vector	**+**	(const Vector v)	Adds two vectors
[const]	Point	**+**	(const Point p)	Adds a vector and a point
[const]	Vector	**-**	(const Vector v)	Subtract two vectors
[const]	Vector	**-@**		Compute the negative of a vector
[const]	Vector	***/***	(double d)	Division by some divisor
	Vector	***/=***	(double d)	Division by some divisor
[const]	bool	**<**	(const Vector v)	"less" comparison operator
[const]	bool	**==**	(const Vector v)	Equality test operator
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object

[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	double	**abs**		Returns the length of the vector
	void	**assign**	(const Vector other)	Assigns another object to self
[const]	new Vector ptr	**dup**		Creates a copy of self
[const]	unsigned long	**hash**		Computes a hash value
[const]	double	**length**		Returns the length of the vector
[const]	long	**sprod**	(const Vector v)	Computes the scalar product between self and the given vector
[const]	int	**sprod_sign**	(const Vector v)	Computes the scalar product between self and the given vector and returns a value indicating the sign of the product
[const]	double	**sq_abs**		The square length of the vector
[const]	double	**sq_length**		The square length of the vector
[const]	DVector	**to_dtype**	(double dbu = 1)	Converts the vector to a floating-point coordinate vector
[const]	Point	**to_p**		Turns the vector into a point
[const]	string	**to_s**	(double dbu = 0)	String conversion
[const]	long	**vprod**	(const Vector v)	Computes the vector product between self and the given vector
[const]	int	**vprod_sign**	(const Vector v)	Computes the vector product between self and the given vector and returns a value indicating the sign of the product
[const]	int	**x**		Accessor to the x coordinate
	void	**x=**	(int coord)	Write accessor to the x coordinate
[const]	int	**y**		Accessor to the y coordinate
	void	**y=**	(int coord)	Write accessor to the y coordinate

Public static methods and constants

new Vector ptr	**from_s**	(string s)	Creates an object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const Vector v)

Description: Inequality test operator

*****

(1) Signature: *[const]* Vector ***** (double f)

Description: Scaling by some factor

Returns the scaled object. All coordinates are multiplied with the given factor and if necessary rounded.

Python specific notes:
This method also implements '__rmul__'.

(2) Signature: *[const]* long ***** (const Vector v)

Description: Computes the scalar product between self and the given vector

The scalar product of a and b is defined as: vp = ax*bx+ay*by.

Python specific notes:
This method also implements '__rmul__'.

***=**

Signature: Vector ***=** (double f)

Description: Scaling by some factor

Scales object in place. All coordinates are multiplied with the given factor and if necessary rounded.

+

(1) Signature: *[const]* Vector **+** (const Vector v)

Description: Adds two vectors

Adds vector v to self by adding the coordinates.

(2) Signature: *[const]* Point **+** (const Point p)

Description: Adds a vector and a point

Returns the point p shifted by the vector.

-	**Signature**: *[const]* Vector **-** (const Vector v) **Description**: Subtract two vectors Subtract vector v from self by subtracting the coordinates.
-@	**Signature**: *[const]* Vector **-@** **Description**: Compute the negative of a vector Returns a new vector with -x,-y.
/	**Signature**: *[const]* Vector **/** (double d) **Description**: Division by some divisor Returns the scaled object. All coordinates are divided with the given divisor and if necessary rounded.
/=	**Signature**: Vector **/=** (double d) **Description**: Division by some divisor Divides the object in place. All coordinates are divided with the given divisor and if necessary rounded.
<	**Signature**: *[const]* bool **<** (const Vector v) **Description**: "less" comparison operator This operator is provided to establish a sorting order
==	**Signature**: *[const]* bool **==** (const Vector v) **Description**: Equality test operator
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

abs

Signature: *[const]* double **abs**

Description: Returns the length of the vector

'abs' is an alias provided for compatibility with the former point type.

assign

Signature: void **assign** (const Vector other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new Vector ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

from_s

Signature: *[static]* new Vector ptr **from_s** (string s)

Description: Creates an object from a string

Creates the object from a string representation (as returned by to_s)

hash

Signature: *[const]* unsigned long **hash**

Description: Computes a hash value

Returns a hash value for the given vector. This method enables vectors as hash keys.

This method has been introduced in version 0.25.

Python specific notes:
This method is also available as 'hash(object)'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

length

Signature: *[const]* double **length**

Description: Returns the length of the vector

'abs' is an alias provided for compatibility with the former point type.

new

(1) Signature: *[static]* new Vector ptr **new** (const DVector dvector)

Description: Creates an integer coordinate vector from a floating-point coordinate vector

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Vector ptr **new**

Description: Default constructor: creates a null vector with coordinates (0,0)

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Vector ptr **new** (const Point p)

Description: Default constructor: creates a vector from a point

This constructor is equivalent to computing p-point(0,0). This method has been introduced in version 0.25.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Vector ptr **new** (int x, int y)

Description: Constructor for a vector from two coordinate values

Python specific notes:
This method is the default initializer of the object.

sprod

Signature: *[const]* long **sprod** (const Vector v)

Description: Computes the scalar product between self and the given vector

The scalar product of a and b is defined as: vp = ax*bx+ay*by.

sprod_sign

Signature: *[const]* int **sprod_sign** (const Vector v)

Description: Computes the scalar product between self and the given vector and returns a value indicating the sign of the product

Returns: 1 if the scalar product is positive, 0 if it is zero and -1 if it is negative.

sq_abs

Signature: *[const]* double **sq_abs**

Description: The square length of the vector

'sq_abs' is an alias provided for compatibility with the former point type.

sq_length

Signature: *[const]* double **sq_length**

Description: The square length of the vector

'sq_abs' is an alias provided for compatibility with the former point type.

to_dtype

Signature: *[const]* DVector **to_dtype** (double dbu = 1)

Description: Converts the vector to a floating-point coordinate vector

The database unit can be specified to translate the integer-coordinate vector into a floating-point coordinate vector in micron units. The database unit is basically a scaling factor.

to_p

Signature: *[const]* Point **to_p**

Description: Turns the vector into a point

This method returns the point resulting from adding the vector to (0,0). This method has been introduced in version 0.25.

to_s

Signature: *[const]* string **to_s** (double dbu = 0)

Description: String conversion

If a DBU is given, the output units will be micrometers.

The DBU argument has been added in version 0.27.6.

Python specific notes:
This method is also available as 'str(object)'.

vprod

Signature: *[const]* long **vprod** (const Vector v)

Description: Computes the vector product between self and the given vector

The vector product of a and b is defined as: vp = ax*by-ay*bx.

vprod_sign

Signature: *[const]* int **vprod_sign** (const Vector v)

Description: Computes the vector product between self and the given vector and returns a value indicating the sign of the product

Returns: 1 if the vector product is positive, 0 if it is zero and -1 if it is negative.

x

Signature: *[const]* int **x**

Description: Accessor to the x coordinate

Python specific notes:
The object exposes a readable attribute 'x'. This is the getter.

x=

Signature: void **x=** (int coord)

Description: Write accessor to the x coordinate

Python specific notes:
The object exposes a writable attribute 'x'. This is the setter.

y

Signature: *[const]* int **y**

Description: Accessor to the y coordinate

Python specific notes:
The object exposes a readable attribute 'y'. This is the getter.

y=

Signature: void **y=** (int coord)

Description: Write accessor to the y coordinate

Python specific notes:
The object exposes a writable attribute 'y'. This is the setter.

4.114. API reference - Class LayoutDiff

Notation used in Ruby API documentation

Module: db

Description: The layout compare tool

The layout compare tool is a facility to quickly compare layouts and derive events that give details about the differences. The events are basically emitted following a certain order:

- General configuration events (database units, layers ...)
- on_begin_cell
- on_begin_inst_differences (if the instances differ)
- details about instance differences (if Verbose flag is given)
- on_end_inst_differences (if the instances differ)
- on_begin_layer
- on_begin_polygon_differences (if the polygons differ)
- details about polygon differences (if Verbose flag is given)
- on_end_polygon_differences (if the polygons differ)
- other shape difference events (paths, boxes, ...)
- on_end_layer
- repeated layer event groups
- on_end_cell
- repeated cell event groups

To use the diff facility, create a LayoutDiff object and call the compare_layout or compare_cell method:

```
lya = ... # layout A
lyb = ... # layout B

diff = RBA::LayoutDiff::new
diff.on_polygon_in_a_only do |poly|
  puts "Polygon in A: #{diff.cell_a.name}@#{diff.layer_info_a.to_s}: #{poly.to_s}"
end
diff.on_polygon_in_b_only do |poly|
  puts "Polygon in A: #{diff.cell_b.name}@#{diff.layer_info_b.to_s}: #{poly.to_s}"
end
diff.compare(lya, lyb, RBA::LayoutDiff::Verbose + RBA::LayoutDiff::NoLayerNames)
```

Public constructors

new LayoutDiff ptr	**new**	Creates a new object of this class

Public methods

	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
	void	assign	(const LayoutDiff other)	Assigns another object to self
[const]	const Cell ptr	cell_a		Gets the current cell for the first layout
[const]	const Cell ptr	cell_b		Gets the current cell for the second layout
	bool	compare	(const Layout ptr a, const Layout ptr b, unsigned int flags = 0, int tolerance = 0)	Compares two layouts
	bool	compare	(const Cell ptr a, const Cell ptr b, unsigned int flags = 0, int tolerance = 0)	Compares two cells
[const]	new LayoutDiff ptr	dup		Creates a copy of self
[const]	int	layer_index_a		Gets the current layer for the first layout
[const]	int	layer_index_b		Gets the current layer for the second layout
[const]	LayerInfo	layer_info_a		Gets the current layer properties for the first layout
[const]	LayerInfo	layer_info_b		Gets the current layer properties for the second layout
[const]	const Layout ptr	layout_a		Gets the first layout the difference detector runs on
[const]	const Layout ptr	layout_b		Gets the second layout the difference detector runs on
[signal]	void	on_bbox_differs	(const Box ba, const Box bb)	This signal indicates a difference in the bounding boxes of two cells

[signal]	void	**on_begin_box_differences**		This signal indicates differences in the boxes on the current layer
[signal]	void	**on_begin_cell**	(const Cell ptr ca, const Cell ptr cb)	This signal indicates the sequence of events for a cell pair
[signal]	void	**on_begin_edge_differences**		This signal indicates differences in the edges on the current layer
[signal]	void	**on_begin_edge_pair_dif**		This signal indicates differences in the edge pairs on the current layer
[signal]	void	**on_begin_inst_differences**		This signal indicates differences in the cell instances
[signal]	void	**on_begin_layer**	(const LayerInfo layer, int layer_index_a, int layer_index_b)	This signal indicates differences on the given layer
[signal]	void	**on_begin_path_differences**		This signal indicates differences in the paths on the current layer
[signal]	void	**on_begin_polygon_diffe**		This signal indicates differences in the polygons on the current layer
[signal]	void	**on_begin_text_differences**		This signal indicates differences in the texts on the current layer
[signal]	void	**on_box_in_a_only**	(const Box anotb, unsigned long prop_id)	This signal indicates a box that is present in the first layout only
[signal]	void	**on_box_in_b_only**	(const Box bnota, unsigned long prop_id)	This signal indicates a box that is present in the second layout only
[signal]	void	**on_cell_in_a_only**	(const Cell ptr c)	This signal indicates that the given cell is only present in the first layout
[signal]	void	**on_cell_in_b_only**	(const Cell ptr c)	This signal indicates that the given cell is only present in the second layout
[signal]	void	**on_cell_meta_info_differ**	(string name, variant a, variant b)	This signal indicates that meta info between the current cells differs
[signal]	void	**on_cell_name_differs**	(const Cell ptr ca, const Cell ptr cb)	This signal indicates a difference in the cell names
[signal]	void	**on_dbu_differs**	(double dbu_a, double dbu_b)	This signal indicates a difference in the database units of the layouts
[signal]	void	**on_edge_in_a_only**	(const Edge anotb, unsigned long prop_id)	This signal indicates an edge that is present in the first layout only
[signal]	void	**on_edge_in_b_only**	(const Edge bnota,	This signal indicates an edge that is present in the second layout only

			unsigned long prop_id)	
[signal]	void	**on_edge_pair_in_a_only**	(const EdgePair anotb, unsigned long prop_id)	This signal indicates an edge pair that is present in the first layout only
[signal]	void	**on_edge_pair_in_b_only**	(const EdgePair bnota, unsigned long prop_id)	This signal indicates an edge pair that is present in the second layout only
[signal]	void	**on_end_box_differences**		This signal indicates the end of sequence of box differences
[signal]	void	**on_end_cell**		This signal indicates the end of a sequence of signals for a specific cell
[signal]	void	**on_end_edge_differences**		This signal indicates the end of sequence of edge differences
[signal]	void	**on_end_edge_pair_diffe**		This signal indicates the end of sequence of edge pair differences
[signal]	void	**on_end_inst_differences**		This signal finishes a sequence of detailed instance difference events
[signal]	void	**on_end_layer**		This signal indicates the end of a sequence of signals for a specific layer
[signal]	void	**on_end_path_differences**		This signal indicates the end of sequence of path differences
[signal]	void	**on_end_polygon_differe**		This signal indicates the end of sequence of polygon differences
[signal]	void	**on_end_text_differences**		This signal indicates the end of sequence of text differences
[signal]	void	**on_instance_in_a_only**	(const CellInstArray anotb, unsigned long prop_id)	This signal indicates an instance that is present only in the first layout
[signal]	void	**on_instance_in_b_only**	(const CellInstArray bnota, unsigned long prop_id)	This signal indicates an instance that is present only in the second layout
[signal]	void	**on_layer_in_a_only**	(const LayerInfo a)	This signal indicates a layer that is present only in the first layout
[signal]	void	**on_layer_in_b_only**	(const LayerInfo b)	This signal indicates a layer that is present only in the second layout

[signal]	void	**on_layer_name_differs**	(const LayerInfo a, const LayerInfo b)	This signal indicates a difference in the layer names
[signal]	void	**on_layout_meta_info_differs**	(string name, variant a, variant b)	This signal indicates that global meta info differs
[signal]	void	**on_path_in_a_only**	(const Path anotb, unsigned long prop_id)	This signal indicates a path that is present in the first layout only
[signal]	void	**on_path_in_b_only**	(const Path bnota, unsigned long prop_id)	This signal indicates a path that is present in the second layout only
[signal]	void	**on_per_layer_bbox_diffe**	(const Box ba, const Box bb)	This signal indicates differences in the per-layer bounding boxes of the current cell
[signal]	void	**on_polygon_in_a_only**	(const Polygon anotb, unsigned long prop_id)	This signal indicates a polygon that is present in the first layout only
[signal]	void	**on_polygon_in_b_only**	(const Polygon bnota, unsigned long prop_id)	This signal indicates a polygon that is present in the second layout only
[signal]	void	**on_text_in_a_only**	(const Text anotb, unsigned long prop_id)	This signal indicates a text that is present in the first layout only
[signal]	void	**on_text_in_b_only**	(const Text bnota, unsigned long prop_id)	This signal indicates a text that is present in the second layout only

Public static methods and constants

[static,const]	unsigned int	**BoxesAsPolygons**	Compare boxes to polygons
[static,const]	unsigned int	**DontSummarizeMissingLayers**	Don't summarize missing layers
[static,const]	unsigned int	**FlattenArrayInsts**	Compare array instances instance by instance
[static,const]	unsigned int	**IgnoreDuplicates**	Ignore duplicate instances or shapes
[static,const]	unsigned int	**NoLayerNames**	Do not compare layer names
[static,const]	unsigned int	**NoProperties**	Ignore properties
[static,const]	unsigned int	**NoTextDetails**	Ignore text details (font, size, presentation)
[static,const]	unsigned int	**NoTextOrientation**	Ignore text orientation
[static,const]	unsigned int	**PathsAsPolygons**	Compare paths to polygons

[static,const]	unsigned int	**Silent**	Silent compare - just report whether the layouts are identical
[static,const]	unsigned int	**SmartCellMapping**	Derive smart cell mapping instead of name mapping (available only if top cells are specified)
[static,const]	unsigned int	**Verbose**	Enables verbose mode (gives details about the differences)
[static,const]	unsigned int	**WithMetaInfo**	Ignore meta info

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

BoxesAsPolygons

Signature: *[static,const]* unsigned int **BoxesAsPolygons**

Description: Compare boxes to polygons

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set.

Python specific notes:
The object exposes a readable attribute 'BoxesAsPolygons'. This is the getter.

DontSummarizeMissingLayers

Signature: *[static,const]* unsigned int **DontSummarizeMissingLayers**

Description: Don't summarize missing layers

If this mode is present, missing layers are treated as empty ones and every shape on the other layer will be reported as difference.

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set.

Python specific notes:
The object exposes a readable attribute 'DontSummarizeMissingLayers'. This is the getter.

FlattenArrayInsts

Signature: *[static,const]* unsigned int **FlattenArrayInsts**

Description: Compare array instances instance by instance

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set.

Python specific notes:
The object exposes a readable attribute 'FlattenArrayInsts'. This is the getter.

IgnoreDuplicates

Signature: *[static,const]* unsigned int **IgnoreDuplicates**

Description: Ignore duplicate instances or shapes

With this option present, duplicate instances or shapes are ignored and duplication does not count as a difference.

This option has been introduced in version 0.28.9.

Python specific notes:
The object exposes a readable attribute 'IgnoreDuplicates'. This is the getter.

NoLayerNames

Signature: *[static,const]* unsigned int **NoLayerNames**

Description: Do not compare layer names

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set.

Python specific notes:
The object exposes a readable attribute 'NoLayerNames'. This is the getter.

NoProperties

Signature: *[static,const]* unsigned int **NoProperties**

Description: Ignore properties

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set.

Python specific notes:
The object exposes a readable attribute 'NoProperties'. This is the getter.

NoTextDetails

Signature: *[static,const]* unsigned int **NoTextDetails**

Description: Ignore text details (font, size, presentation)

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set.

Python specific notes:
The object exposes a readable attribute 'NoTextDetails'. This is the getter.

NoTextOrientation

Signature: *[static,const]* unsigned int **NoTextOrientation**

Description: Ignore text orientation

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set.

Python specific notes:
The object exposes a readable attribute 'NoTextOrientation'. This is the getter.

PathsAsPolygons

Signature: *[static,const]* unsigned int **PathsAsPolygons**

Description: Compare paths to polygons

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set.

Python specific notes:
The object exposes a readable attribute 'PathsAsPolygons'. This is the getter.

Silent

Signature: *[static,const]* unsigned int **Silent**

Description: Silent compare - just report whether the layouts are identical

Silent mode will not issue any signals, but instead the return value of the LayoutDiff#compare method will indicate whether the layouts are identical. In silent mode, the compare method will return immediately once a difference has been encountered so that mode may be much faster than the full compare.

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set.

Python specific notes:
The object exposes a readable attribute 'Silent'. This is the getter.

SmartCellMapping

Signature: *[static,const]* unsigned int **SmartCellMapping**

Description: Derive smart cell mapping instead of name mapping (available only if top cells are specified)

Smart cell mapping is only effective currently when cells are compared (with LayoutDiff#compare with cells instead of layout objects).

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set.

Python specific notes:
The object exposes a readable attribute 'SmartCellMapping'. This is the getter.

Verbose

Signature: *[static,const]* unsigned int **Verbose**

Description: Enables verbose mode (gives details about the differences)

See the event descriptions for details about the differences in verbose and non-verbose mode.

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set.

Python specific notes:
The object exposes a readable attribute 'Verbose'. This is the getter.

WithMetaInfo

Signature: *[static,const]* unsigned int **WithMetaInfo**

Description: Ignore meta info

This constant can be used for the flags parameter of compare_layouts and compare_cells. It can be combined with other constants to form a flag set. If present, this option tells the compare algorithm to include persisted meta info in the compare.

This flag has been introduced in version 0.28.16.

Python specific notes:
The object exposes a readable attribute 'WithMetaInfo'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const LayoutDiff other)

Description: Assigns another object to self

cell_a

Signature: *[const]* const Cell ptr **cell_a**

Description: Gets the current cell for the first layout

This attribute is the current cell and is set after on_begin_cell and reset after on_end_cell.

cell_b

Signature: *[const]* const Cell ptr **cell_b**

Description: Gets the current cell for the second layout

This attribute is the current cell and is set after on_begin_cell and reset after on_end_cell.

compare

(1) Signature: bool **compare** (const Layout ptr a, const Layout ptr b, unsigned int flags = 0, int tolerance = 0)

Description: Compares two layouts

a:	The first input layout
b:	The second input layout
flags:	Flags to use for the comparison
tolerance:	A coordinate tolerance to apply (0: exact match, 1: one DBU tolerance is allowed ...)
Returns:	True, if the layouts are identical

Compares layer definitions, cells, instances and shapes and properties. Cells are identified by name. Only layers with valid layer and datatype are compared. Several flags can be specified as a bitwise or combination of the constants.

(2) Signature: bool **compare** (const Cell ptr a, const Cell ptr b, unsigned int flags = 0, int tolerance = 0)

Description: Compares two cells

a:	The first top cell
b:	The second top cell
flags:	Flags to use for the comparison
tolerance:	A coordinate tolerance to apply (0: exact match, 1: one DBU tolerance is allowed ...)
Returns:	True, if the cells are identical

Compares layer definitions, cells, instances and shapes and properties of two layout hierarchies starting from the given cells. Cells are identified by name. Only layers with valid layer and datatype are compared. Several flags can be specified as a bitwise or combination of the constants.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new LayoutDiff ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

layer_index_a	**Signature**: *[const]* int **layer_index_a** **Description**: Gets the current layer for the first layout This attribute is the current cell and is set after on_begin_layer and reset after on_end_layer.
layer_index_b	**Signature**: *[const]* int **layer_index_b** **Description**: Gets the current layer for the second layout This attribute is the current cell and is set after on_begin_layer and reset after on_end_layer.
layer_info_a	**Signature**: *[const]* LayerInfo **layer_info_a** **Description**: Gets the current layer properties for the first layout This attribute is the current cell and is set after on_begin_layer and reset after on_end_layer.
layer_info_b	**Signature**: *[const]* LayerInfo **layer_info_b** **Description**: Gets the current layer properties for the second layout This attribute is the current cell and is set after on_begin_layer and reset after on_end_layer.
layout_a	**Signature**: *[const]* const Layout ptr **layout_a** **Description**: Gets the first layout the difference detector runs on
layout_b	**Signature**: *[const]* const Layout ptr **layout_b** **Description**: Gets the second layout the difference detector runs on
new	**Signature**: *[static]* new LayoutDiff ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
on_bbox_differs	**Signature**: *[signal]* void **on_bbox_differs** (const Box ba, const Box bb) **Description**: This signal indicates a difference in the bounding boxes of two cells This signal is only emitted in non-verbose mode (without Verbose flag) as a summarizing cell property. In verbose mode detailed events will be issued indicating the differences. **Python specific notes:** The object exposes a readable attribute 'on_bbox_differs'. This is the getter. The object exposes a writable attribute 'on_bbox_differs'. This is the setter.
on_begin_box_differences	**Signature**: *[signal]* void **on_begin_box_differences** **Description**: This signal indicates differences in the boxes on the current layer The current layer is indicated by the begin_layer_event signal or can be obtained from the diff object through LayoutDiff#layer_info_a, LayoutDiff#layer_index_a, LayoutDiff#layer_info_b and LayoutDiff#layer_index_b. In verbose mode (see Verbose flag) more signals will be emitted for boxes that are different between the two layouts. **Python specific notes:** The object exposes a readable attribute 'on_begin_box_differences'. This is the getter. The object exposes a writable attribute 'on_begin_box_differences'. This is the setter.

on_begin_cell

Signature: *[signal]* void **on_begin_cell** (const Cell ptr ca, const Cell ptr cb)

Description: This signal indicates the sequence of events for a cell pair

All cell specific events happen between begin_cell_event and end_cell_event signals.

Python specific notes:
The object exposes a readable attribute 'on_begin_cell'. This is the getter.
The object exposes a writable attribute 'on_begin_cell'. This is the setter.

on_begin_edge_differences

Signature: *[signal]* void **on_begin_edge_differences**

Description: This signal indicates differences in the edges on the current layer

The current layer is indicated by the begin_layer_event signal or can be obtained from the diff object through LayoutDiff#layer_info_a, LayoutDiff#layer_index_a, LayoutDiff#layer_info_b and LayoutDiff#layer_index_b. In verbose mode (see Verbose flag) more signals will be emitted for edges that are different between the two layouts.

Python specific notes:
The object exposes a readable attribute 'on_begin_edge_differences'. This is the getter.
The object exposes a writable attribute 'on_begin_edge_differences'. This is the setter.

on_begin_edge_pair_differences

Signature: *[signal]* void **on_begin_edge_pair_differences**

Description: This signal indicates differences in the edge pairs on the current layer

The current layer is indicated by the begin_layer_event signal or can be obtained from the diff object through LayoutDiff#layer_info_a, LayoutDiff#layer_index_a, LayoutDiff#layer_info_b and LayoutDiff#layer_index_b. In verbose mode (see Verbose flag) more signals will be emitted for edge pairs that are different between the two layouts. This event has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'on_begin_edge_pair_differences'. This is the getter.
The object exposes a writable attribute 'on_begin_edge_pair_differences'. This is the setter.

on_begin_inst_differences

Signature: *[signal]* void **on_begin_inst_differences**

Description: This signal indicates differences in the cell instances

In verbose mode (see Verbose) more events will follow that indicate the instances that are present only in the first and second layout (instance_in_a_only_event and instance_in_b_only_event).

Python specific notes:
The object exposes a readable attribute 'on_begin_inst_differences'. This is the getter.
The object exposes a writable attribute 'on_begin_inst_differences'. This is the setter.

on_begin_layer

Signature: *[signal]* void **on_begin_layer** (const LayerInfo layer, int layer_index_a, int layer_index_b)

Description: This signal indicates differences on the given layer

In verbose mode (see Verbose) more events will follow that indicate the instances that are present only in the first and second layout (polygon_in_a_only_event, polygon_in_b_only_event and similar).

Python specific notes:
The object exposes a readable attribute 'on_begin_layer'. This is the getter.
The object exposes a writable attribute 'on_begin_layer'. This is the setter.

on_begin_path_differences

Signature: *[signal]* void **on_begin_path_differences**

Description: This signal indicates differences in the paths on the current layer

The current layer is indicated by the begin_layer_event signal or can be obtained from the diff object through LayoutDiff#layer_info_a, LayoutDiff#layer_index_a, LayoutDiff#layer_info_b and

	LayoutDiff#layer_index_b. In verbose mode (see Verbose flag) more signals will be emitted for paths that are different between the two layouts. **Python specific notes:** The object exposes a readable attribute 'on_begin_path_differences'. This is the getter. The object exposes a writable attribute 'on_begin_path_differences'. This is the setter.
on_begin_polygon_differences	**Signature**: *[signal]* void **on_begin_polygon_differences** **Description**: This signal indicates differences in the polygons on the current layer The current layer is indicated by the begin_layer_event signal or can be obtained from the diff object through LayoutDiff#layer_info_a, LayoutDiff#layer_index_a, LayoutDiff#layer_info_b and LayoutDiff#layer_index_b. In verbose mode (see Verbose flag) more signals will be emitted for polygons that are different between the two layouts. **Python specific notes:** The object exposes a readable attribute 'on_begin_polygon_differences'. This is the getter. The object exposes a writable attribute 'on_begin_polygon_differences'. This is the setter.
on_begin_text_differences	**Signature**: *[signal]* void **on_begin_text_differences** **Description**: This signal indicates differences in the texts on the current layer The current layer is indicated by the begin_layer_event signal or can be obtained from the diff object through LayoutDiff#layer_info_a, LayoutDiff#layer_index_a, LayoutDiff#layer_info_b and LayoutDiff#layer_index_b. In verbose mode (see Verbose flag) more signals will be emitted for texts that are different between the two layouts. **Python specific notes:** The object exposes a readable attribute 'on_begin_text_differences'. This is the getter. The object exposes a writable attribute 'on_begin_text_differences'. This is the setter.
on_box_in_a_only	**Signature**: *[signal]* void **on_box_in_a_only** (const Box anotb, unsigned long prop_id) **Description**: This signal indicates a box that is present in the first layout only **Python specific notes:** The object exposes a readable attribute 'on_box_in_a_only'. This is the getter. The object exposes a writable attribute 'on_box_in_a_only'. This is the setter.
on_box_in_b_only	**Signature**: *[signal]* void **on_box_in_b_only** (const Box bnota, unsigned long prop_id) **Description**: This signal indicates a box that is present in the second layout only **Python specific notes:** The object exposes a readable attribute 'on_box_in_b_only'. This is the getter. The object exposes a writable attribute 'on_box_in_b_only'. This is the setter.
on_cell_in_a_only	**Signature**: *[signal]* void **on_cell_in_a_only** (const Cell ptr c) **Description**: This signal indicates that the given cell is only present in the first layout **Python specific notes:** The object exposes a readable attribute 'on_cell_in_a_only'. This is the getter. The object exposes a writable attribute 'on_cell_in_a_only'. This is the setter.
on_cell_in_b_only	**Signature**: *[signal]* void **on_cell_in_b_only** (const Cell ptr c) **Description**: This signal indicates that the given cell is only present in the second layout **Python specific notes:** The object exposes a readable attribute 'on_cell_in_b_only'. This is the getter. The object exposes a writable attribute 'on_cell_in_b_only'. This is the setter.

on_cell_meta_info_differs

Signature: *[signal]* void **on_cell_meta_info_differs** (string name, variant a, variant b)

Description: This signal indicates that meta info between the current cells differs

Meta information is only compared when WithMetaInfo is added to the compare flags. 'a' and 'b' are the values for the first and second layout. 'nil' is passed to these values to indicate missing meta information on one side.

This event has been added in version 0.28.16.

Python specific notes:
The object exposes a readable attribute 'on_cell_meta_info_differs'. This is the getter.
The object exposes a writable attribute 'on_cell_meta_info_differs'. This is the setter.

on_cell_name_differs

Signature: *[signal]* void **on_cell_name_differs** (const Cell ptr ca, const Cell ptr cb)

Description: This signal indicates a difference in the cell names

This signal is emitted in 'smart cell mapping' mode (see SmartCellMapping) if two cells are considered identical, but have different names.

Python specific notes:
The object exposes a readable attribute 'on_cell_name_differs'. This is the getter.
The object exposes a writable attribute 'on_cell_name_differs'. This is the setter.

on_dbu_differs

Signature: *[signal]* void **on_dbu_differs** (double dbu_a, double dbu_b)

Description: This signal indicates a difference in the database units of the layouts

Python specific notes:
The object exposes a readable attribute 'on_dbu_differs'. This is the getter.
The object exposes a writable attribute 'on_dbu_differs'. This is the setter.

on_edge_in_a_only

Signature: *[signal]* void **on_edge_in_a_only** (const Edge anotb, unsigned long prop_id)

Description: This signal indicates an edge that is present in the first layout only

Python specific notes:
The object exposes a readable attribute 'on_edge_in_a_only'. This is the getter.
The object exposes a writable attribute 'on_edge_in_a_only'. This is the setter.

on_edge_in_b_only

Signature: *[signal]* void **on_edge_in_b_only** (const Edge bnota, unsigned long prop_id)

Description: This signal indicates an edge that is present in the second layout only

Python specific notes:
The object exposes a readable attribute 'on_edge_in_b_only'. This is the getter.
The object exposes a writable attribute 'on_edge_in_b_only'. This is the setter.

on_edge_pair_in_a_only

Signature: *[signal]* void **on_edge_pair_in_a_only** (const EdgePair anotb, unsigned long prop_id)

Description: This signal indicates an edge pair that is present in the first layout only

This event has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'on_edge_pair_in_a_only'. This is the getter.
The object exposes a writable attribute 'on_edge_pair_in_a_only'. This is the setter.

on_edge_pair_in_b_only

Signature: *[signal]* void **on_edge_pair_in_b_only** (const EdgePair bnota, unsigned long prop_id)

Description: This signal indicates an edge pair that is present in the second layout only

This event has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'on_edge_pair_in_b_only'. This is the getter.
The object exposes a writable attribute 'on_edge_pair_in_b_only'. This is the setter.

on_end_box_differences

Signature: *[signal]* void **on_end_box_differences**

Description: This signal indicates the end of sequence of box differences

Python specific notes:
The object exposes a readable attribute 'on_end_box_differences'. This is the getter.
The object exposes a writable attribute 'on_end_box_differences'. This is the setter.

on_end_cell

Signature: *[signal]* void **on_end_cell**

Description: This signal indicates the end of a sequence of signals for a specific cell

Python specific notes:
The object exposes a readable attribute 'on_end_cell'. This is the getter.
The object exposes a writable attribute 'on_end_cell'. This is the setter.

on_end_edge_differences

Signature: *[signal]* void **on_end_edge_differences**

Description: This signal indicates the end of sequence of edge differences

Python specific notes:
The object exposes a readable attribute 'on_end_edge_differences'. This is the getter.
The object exposes a writable attribute 'on_end_edge_differences'. This is the setter.

on_end_edge_pair_differences

Signature: *[signal]* void **on_end_edge_pair_differences**

Description: This signal indicates the end of sequence of edge pair differences

This event has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'on_end_edge_pair_differences'. This is the getter.
The object exposes a writable attribute 'on_end_edge_pair_differences'. This is the setter.

on_end_inst_differences

Signature: *[signal]* void **on_end_inst_differences**

Description: This signal finishes a sequence of detailed instance difference events

Python specific notes:
The object exposes a readable attribute 'on_end_inst_differences'. This is the getter.
The object exposes a writable attribute 'on_end_inst_differences'. This is the setter.

on_end_layer

Signature: *[signal]* void **on_end_layer**

Description: This signal indicates the end of a sequence of signals for a specific layer

Python specific notes:
The object exposes a readable attribute 'on_end_layer'. This is the getter.
The object exposes a writable attribute 'on_end_layer'. This is the setter.

on_end_path_differences

Signature: *[signal]* void **on_end_path_differences**

Description: This signal indicates the end of sequence of path differences

Python specific notes:
The object exposes a readable attribute 'on_end_path_differences'. This is the getter.
The object exposes a writable attribute 'on_end_path_differences'. This is the setter.

on_end_polygon_differences

Signature: *[signal]* void **on_end_polygon_differences**

Description: This signal indicates the end of sequence of polygon differences

Python specific notes:
The object exposes a readable attribute 'on_end_polygon_differences'. This is the getter.
The object exposes a writable attribute 'on_end_polygon_differences'. This is the setter.

on_end_text_differences

Signature: *[signal]* void **on_end_text_differences**

Description: This signal indicates the end of sequence of text differences

Python specific notes:
The object exposes a readable attribute 'on_end_text_differences'. This is the getter.
The object exposes a writable attribute 'on_end_text_differences'. This is the setter.

on_instance_in_a_only

Signature: *[signal]* void **on_instance_in_a_only** (const CellInstArray anotb, unsigned long prop_id)

Description: This signal indicates an instance that is present only in the first layout

This event is only emitted in verbose mode (Verbose flag).

Python specific notes:
The object exposes a readable attribute 'on_instance_in_a_only'. This is the getter.
The object exposes a writable attribute 'on_instance_in_a_only'. This is the setter.

on_instance_in_b_only

Signature: *[signal]* void **on_instance_in_b_only** (const CellInstArray bnota, unsigned long prop_id)

Description: This signal indicates an instance that is present only in the second layout

This event is only emitted in verbose mode (Verbose flag).

Python specific notes:
The object exposes a readable attribute 'on_instance_in_b_only'. This is the getter.
The object exposes a writable attribute 'on_instance_in_b_only'. This is the setter.

on_layer_in_a_only

Signature: *[signal]* void **on_layer_in_a_only** (const LayerInfo a)

Description: This signal indicates a layer that is present only in the first layout

Python specific notes:
The object exposes a readable attribute 'on_layer_in_a_only'. This is the getter.
The object exposes a writable attribute 'on_layer_in_a_only'. This is the setter.

on_layer_in_b_only

Signature: *[signal]* void **on_layer_in_b_only** (const LayerInfo b)

Description: This signal indicates a layer that is present only in the second layout

Python specific notes:
The object exposes a readable attribute 'on_layer_in_b_only'. This is the getter.
The object exposes a writable attribute 'on_layer_in_b_only'. This is the setter.

on_layer_name_differs

Signature: *[signal]* void **on_layer_name_differs** (const LayerInfo a, const LayerInfo b)

Description: This signal indicates a difference in the layer names

Python specific notes:
The object exposes a readable attribute 'on_layer_name_differs'. This is the getter.
The object exposes a writable attribute 'on_layer_name_differs'. This is the setter.

on_layout_meta_info_differs

Signature: *[signal]* void **on_layout_meta_info_differs** (string name, variant a, variant b)

Description: This signal indicates that global meta info differs

	Meta information is only compared when WithMetaInfo is added to the compare flags. 'a' and 'b' are the values for the first and second layout. 'nil' is passed to these values to indicate missing meta information on one side. This event has been added in version 0.28.16. **Python specific notes:** The object exposes a readable attribute 'on_layout_meta_info_differs'. This is the getter. The object exposes a writable attribute 'on_layout_meta_info_differs'. This is the setter.
on_path_in_a_only	**Signature**: *[signal]* void **on_path_in_a_only** (const Path anotb, unsigned long prop_id) **Description**: This signal indicates a path that is present in the first layout only **Python specific notes:** The object exposes a readable attribute 'on_path_in_a_only'. This is the getter. The object exposes a writable attribute 'on_path_in_a_only'. This is the setter.
on_path_in_b_only	**Signature**: *[signal]* void **on_path_in_b_only** (const Path bnota, unsigned long prop_id) **Description**: This signal indicates a path that is present in the second layout only **Python specific notes:** The object exposes a readable attribute 'on_path_in_b_only'. This is the getter. The object exposes a writable attribute 'on_path_in_b_only'. This is the setter.
on_per_layer_bbox_differs	**Signature**: *[signal]* void **on_per_layer_bbox_differs** (const Box ba, const Box bb) **Description**: This signal indicates differences in the per-layer bounding boxes of the current cell **Python specific notes:** The object exposes a readable attribute 'on_per_layer_bbox_differs'. This is the getter. The object exposes a writable attribute 'on_per_layer_bbox_differs'. This is the setter.
on_polygon_in_a_only	**Signature**: *[signal]* void **on_polygon_in_a_only** (const Polygon anotb, unsigned long prop_id) **Description**: This signal indicates a polygon that is present in the first layout only **Python specific notes:** The object exposes a readable attribute 'on_polygon_in_a_only'. This is the getter. The object exposes a writable attribute 'on_polygon_in_a_only'. This is the setter.
on_polygon_in_b_only	**Signature**: *[signal]* void **on_polygon_in_b_only** (const Polygon bnota, unsigned long prop_id) **Description**: This signal indicates a polygon that is present in the second layout only **Python specific notes:** The object exposes a readable attribute 'on_polygon_in_b_only'. This is the getter. The object exposes a writable attribute 'on_polygon_in_b_only'. This is the setter.
on_text_in_a_only	**Signature**: *[signal]* void **on_text_in_a_only** (const Text anotb, unsigned long prop_id) **Description**: This signal indicates a text that is present in the first layout only **Python specific notes:** The object exposes a readable attribute 'on_text_in_a_only'. This is the getter. The object exposes a writable attribute 'on_text_in_a_only'. This is the setter.
on_text_in_b_only	**Signature**: *[signal]* void **on_text_in_b_only** (const Text bnota, unsigned long prop_id) **Description**: This signal indicates a text that is present in the second layout only **Python specific notes:** The object exposes a readable attribute 'on_text_in_b_only'. This is the getter.

The object exposes a writable attribute 'on_text_in_b_only'. This is the setter.

4.115. API reference - Class TextGenerator

Notation used in Ruby API documentation

Module: db

Description: A text generator class

A text generator is basically a way to produce human-readable text for labelling layouts. It's similar to the Basic.TEXT PCell, but more convenient to use in a scripting context.

Generators can be constructed from font files (or resources) or one of the registered generators can be used.

To create a generator from a font file proceed this way:

```
gen = RBA::TextGenerator::new
gen.load_from_file("myfont.gds")
region = gen.text("A TEXT", 0.001)
```

This code produces a RBA::Region with a database unit of 0.001 micron. This region can be fed into a Shapes container to place it into a cell for example.

By convention the font files must have two to three layers:

- 1/0 for the actual data
- 2/0 for the borders
- 3/0 for an optional additional background

Currently, all glyphs must be bottom-left aligned at 0, 0. The border must be drawn in at least one glyph cell. The border is taken as the overall bbox of all borders.

The glyph cells must be named with a single character or "nnn" where "d" is the ASCII code of the character (i.e. "032" for space). Allowed ASCII codes are 32 through 127. If a lower-case "a" character is defined, lower-case letters are supported. Otherwise, lowercase letters are mapped to uppercase letters.

Undefined characters are left blank in the output.

A comment cell can be defined ("COMMENT") which must hold one text in layer 1 stating the comment, and additional descriptions such as line width:

- "line_width=<x>": Specifies the intended line width in micron units
- "design_grid=<x>": Specifies the intended design grid in micron units
- any other text: The description string

Generators can be picked form a list of predefined generator. See generators, default_generator and generator_by_name for picking a generator from the list.

This class has been introduced in version 0.25.

Public constructors

new TextGenerator ptr	**new**	Creates a new object of this class

Public methods

void	**_create**	Ensures the C++ object is created
void	**_destroy**	Explicitly destroys the object

[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const TextGenerato other)	Assigns another object to self
[const]	Box	**background**		Gets the background rectangle of each glyph in the generator's database units
[const]	DBox	**dbackground**		Gets the background rectangle in micron units
[const]	double	**dbu**		Gets the basic database unit the design of the glyphs was made
[const]	double	**ddesign_grid**		Gets the design grid of the glyphs in micron units
[const]	string	**description**		Gets the description text of the generator
[const]	int	**design_grid**		Gets the design grid of the glyphs in the generator's database units
[const]	double	**dheight**		Gets the design height of the glyphs in micron units
[const]	double	**dline_width**		Gets the line width of the glyphs in micron units
[const]	new TextGenerator ptr	**dup**		Creates a copy of self
[const]	double	**dwidth**		Gets the design width of the glyphs in micron units
[const]	Region	**glyph**	(char char)	Gets the glyph of the given character as a region
[const]	int	**height**		Gets the design height of the glyphs in the generator's database units
[const]	int	**line_width**		Gets the line width of the glyphs in the generator's database units
	void	**load_from_file**	(string path)	Loads the given file into the generator
	void	**load_from_resource**	(string resource_path)	Loads the given resource data (as layout data) into the generator
[const]	string	**name**		Gets the name of the generator
[const]	Region	**text**	(string text, double target_dbu,	Gets the rendered text as a region

			double mag = 1, bool inv = false, double bias = 0, double char_spacing = 0, double line_spacing = 0)	
[const]	int	**width**		Gets the design height of the glyphs in the generator's database units

Public static methods and constants

	const TextGenerator ptr	**default_generator**		Gets the default text generator (a standard font)
	string[]	**font_paths**		Gets the paths where to look for font files
	const TextGenerator ptr	**generator_by_name**	(string name)	Gets the text generator for a given name
	const TextGenerator ptr[]	**generators**		Gets the generators registered in the system
	void	**set_font_paths**	(string[] paths)	Sets the paths where to look for font files

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const TextGenerator other)

Description: Assigns another object to self

background

Signature: *[const]* Box **background**

Description: Gets the background rectangle of each glyph in the generator's database units

The background rectangle is the one that is used as background for inverted rendering. A version that delivers this value in micrometer units is dbackground.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

dbackground

Signature: *[const]* DBox **dbackground**

Description: Gets the background rectangle in micron units

The background rectangle is the one that is used as background for inverted rendering.

dbu

Signature: *[const]* double **dbu**

Description: Gets the basic database unit the design of the glyphs was made

This database unit the basic resolution of the glyphs.

ddesign_grid

Signature: *[const]* double **ddesign_grid**

Description: Gets the design grid of the glyphs in micron units

The design grid is the basic grid used when designing the glyphs. In most cases this grid is bigger than the database unit.

default_generator

Signature: *[static]* const TextGenerator ptr **default_generator**

Description: Gets the default text generator (a standard font)

This method delivers the default generator or nil if no such generator is installed.

description

Signature: *[const]* string **description**

Description: Gets the description text of the generator

The generator's description text is a human-readable text that is used to identify the generator (aka 'font') in user interfaces.

design_grid

Signature: *[const]* int **design_grid**

Description: Gets the design grid of the glyphs in the generator's database units

The design grid is the basic grid used when designing the glyphs. In most cases this grid is bigger than the database unit. A version that delivers this value in micrometer units is ddesign_grid.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dheight

Signature: *[const]* double **dheight**

Description: Gets the design height of the glyphs in micron units

The height is the height of the rectangle occupied by each character.

dline_width

Signature: *[const]* double **dline_width**

Description: Gets the line width of the glyphs in micron units

The line width is the intended (not necessarily precisely) line width of typical character lines (such as the bar of an 'I').

dup

Signature: *[const]* new TextGenerator ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

dwidth

Signature: *[const]* double **dwidth**

Description: Gets the design width of the glyphs in micron units

The width is the width of the rectangle occupied by each character.

font_paths

Signature: *[static]* string[] **font_paths**

Description: Gets the paths where to look for font files

See set_font_paths for a description of this function.

This method has been introduced in version 0.27.4.

generator_by_name

Signature: *[static]* const TextGenerator ptr **generator_by_name** (string name)

Description: Gets the text generator for a given name

This method delivers the generator with the given name or nil if no such generator is registered.

generators

Signature: *[static]* const TextGenerator ptr[] **generators**

Description: Gets the generators registered in the system

This method delivers a list of generator objects that can be used to create texts.

glyph

Signature: *[const]* Region **glyph** (char char)

Description: Gets the glyph of the given character as a region

The region represents the glyph's outline and is delivered in the generator's database units .A more elaborate way to getting the text's outline is text.

height

Signature: *[const]* int **height**

Description: Gets the design height of the glyphs in the generator's database units

The height is the height of the rectangle occupied by each character. A version that delivers this value in micrometer units is dheight.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

line_width

Signature: *[const]* int **line_width**

Description: Gets the line width of the glyphs in the generator's database units

The line width is the intended (not necessarily precisely) line width of typical character lines (such as the bar of an 'I'). A version that delivers this value in micrometer units is dline_width.

load_from_file

Signature: void **load_from_file** (string path)

Description: Loads the given file into the generator

See the description of the class how the layout data is read.

load_from_resource

Signature: void **load_from_resource** (string resource_path)

Description: Loads the given resource data (as layout data) into the generator

The resource path has to start with a colon, i.e. ':/my/resource.gds'. See the description of the class how the layout data is read.

name

Signature: *[const]* string **name**

Description: Gets the name of the generator

The generator's name is the basic key by which the generator is identified.

new

Signature: *[static]* new TextGenerator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

set_font_paths

Signature: *[static]* void **set_font_paths** (string[] paths)

Description: Sets the paths where to look for font files

This function sets the paths where to look for font files. After setting such a path, each font found will render a specific generator. The generator can be found under the font file's name. As the text generator is also the basis for the Basic.TEXT PCell, using this function also allows configuring custom fonts for this library cell.

This method has been introduced in version 0.27.4.

text

Signature: *[const]* Region **text** (string text, double target_dbu, double mag = 1, bool inv = false, double bias = 0, double char_spacing = 0, double line_spacing = 0)

Description: Gets the rendered text as a region

text:	The text string
target_dbu:	The database unit for which to produce the text
mag:	The magnification (1.0 for original size)
inv:	inverted rendering: if true, the glyphs are rendered inverse with the background box as the outer bounding box
bias:	An additional bias to be applied (happens before inversion, can be negative)
char_spacing:	Additional space between characters (in micron units)
line_spacing:	Additional space between lines (in micron units)

Various options can be specified to control the appearance of the text. See the description of the parameters. It's important to specify the target database unit in target_dbu to indicate what database unit shall be used to create the output for.

width

Signature: *[const]* int **width**

Description: Gets the design height of the glyphs in the generator's database units

The width is the width of the rectangle occupied by each character. A version that delivers this value in micrometer units is dwidth.

4.116. API reference - Class NetlistObject

Notation used in Ruby API documentation

Module: db

Description: The base class for some netlist objects.

The main purpose of this class is to supply user properties for netlist objects.

This class has been introduced in version 0.26.2

Public constructors

new NetlistObject ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetlistObject other)	Assigns another object to self
[const]	new NetlistObject ptr	**dup**		Creates a copy of self
[const]	variant	**property**	(variant key)	Gets the property value for the given key or nil if there is no value with this key.
[const]	variant[]	**property_keys**		Gets the keys for the properties stored in this object.
	void	**set_property**	(variant key, variant value)	Sets the property value for the given key.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead

[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const NetlistObject other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetlistObject ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new NetlistObject ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

property

Signature: *[const]* variant **property** (variant key)

Description: Gets the property value for the given key or nil if there is no value with this key.

property_keys

Signature: *[const]* variant[] **property_keys**

Description: Gets the keys for the properties stored in this object.

set_property

Signature: void **set_property** (variant key, variant value)

Description: Sets the property value for the given key.

Use a nil value to erase the property with this key.

4.117. API reference - Class Pin

Notation used in Ruby API documentation

Module: db

Description: A pin of a circuit.

Class hierarchy: Pin » NetlistObject

Pin objects are used to describe the outgoing pins of a circuit. To create a new pin of a circuit, use Circuit#create_pin.

This class has been added in version 0.26.

Public methods

	void	**_assign**	(const Pin other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new Pin ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	string	**expanded_name**		Gets the expanded name of the pin.
[const]	unsigned long	**id**		Gets the ID of the pin.
[const]	string	**name**		Gets the name of the pin.

Detailed description

_assign

Signature: void **_assign** (const Pin other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

	Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_dup	**Signature**: *[const]* new Pin ptr **_dup** **Description**: Creates a copy of self
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
expanded_name	**Signature**: *[const]* string **expanded_name** **Description**: Gets the expanded name of the pin. The expanded name is the name or a generic identifier made from the ID if the name is empty.
id	**Signature**: *[const]* unsigned long **id** **Description**: Gets the ID of the pin.
name	**Signature**: *[const]* string **name** **Description**: Gets the name of the pin.

4.118. API reference - Class DeviceReconnectedTerminal

Notation used in Ruby API documentation

Module: db

Description: Describes a terminal rerouting in combined devices.

Combined devices are implemented as a generalization of the device abstract concept in Device. For combined devices, multiple DeviceAbstract references are present. To support different combination schemes, device-to-abstract routing is supported. Parallel combinations will route all outer terminals to corresponding terminals of all device abstracts (because of terminal swapping these may be different ones).

This object describes one route to an abstract's terminal. The device index is 0 for the main device abstract and 1 for the first combined device abstract.

This class has been introduced in version 0.26.

Public constructors

new DeviceReconnectedTerminal ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const DeviceReconne other)	Assigns another object to self
[const]	unsigned long	**device_index**		The device abstract index getter.
	void	**device_index=**	(unsigned long device_index)	The device abstract index setter.
[const]	new DeviceReconnectedTerminal ptr	**dup**		Creates a copy of self
[const]	unsigned long	**other_terminal_id**		The getter for the abstract's connected terminal.
	void	**other_terminal_id=**	(unsigned int other_terminal_id)	The setter for the abstract's connected terminal.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const DeviceReconnectedTerminal other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

device_index

Signature: *[const]* unsigned long **device_index**

Description: The device abstract index getter.

See the class description for details.

Python specific notes:
The object exposes a readable attribute 'device_index'. This is the getter.

device_index=

Signature: void **device_index=** (unsigned long device_index)

Description: The device abstract index setter.

See the class description for details.

Python specific notes:
The object exposes a writable attribute 'device_index'. This is the setter.

dup

Signature: *[const]* new DeviceReconnectedTerminal ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

	Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**Signature**: *[static]* new DeviceReconnectedTerminal ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
other_terminal_id	**Signature**: *[const]* unsigned long **other_terminal_id** **Description**: The getter for the abstract's connected terminal. See the class description for details. **Python specific notes:** The object exposes a readable attribute 'other_terminal_id'. This is the getter.
other_terminal_id=	**Signature**: void **other_terminal_id=** (unsigned int other_terminal_id) **Description**: The setter for the abstract's connected terminal. See the class description for details. **Python specific notes:** The object exposes a writable attribute 'other_terminal_id'. This is the setter.

4.119. API reference - Class DeviceAbstractRef

Notation used in Ruby API documentation

Module: db

Description: Describes an additional device abstract reference for combined devices.

Combined devices are implemented as a generalization of the device abstract concept in Device. For combined devices, multiple DeviceAbstract references are present. This class describes such an additional reference. A reference is a pointer to an abstract plus a transformation by which the abstract is transformed geometrically as compared to the first (initial) abstract.

This class has been introduced in version 0.26.

Public constructors

new DeviceAbstractRef ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const DeviceAbstract other)	Assigns another object to self
[const]	const DeviceAbstract ptr	**device_abstract**		The getter for the device abstract reference.
	void	**device_abstract=**	(const DeviceAbstract ptr device_abstrac	The setter for the device abstract reference.
[const]	new DeviceAbstractRef ptr	**dup**		Creates a copy of self
[const]	DCplxTrans	**trans**		The getter for the relative transformation of the instance.
	void	**trans=**	(const DCplxTrans tr)	The setter for the relative transformation of the instance.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const DeviceAbstractRef other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

device_abstract

Signature: *[const]* const DeviceAbstract ptr **device_abstract**

Description: The getter for the device abstract reference.

See the class description for details.

Python specific notes:
The object exposes a readable attribute 'device_abstract'. This is the getter.

device_abstract=

Signature: void **device_abstract=** (const DeviceAbstract ptr device_abstract)

Description: The setter for the device abstract reference.

See the class description for details.

Python specific notes:
The object exposes a writable attribute 'device_abstract'. This is the setter.

dup

Signature: *[const]* new DeviceAbstractRef ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new DeviceAbstractRef ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

trans

Signature: *[const]* DCplxTrans **trans**

Description: The getter for the relative transformation of the instance.

See the class description for details.

Python specific notes:
The object exposes a readable attribute 'trans'. This is the getter.

trans=

Signature: void **trans=** (const DCplxTrans tr)

Description: The setter for the relative transformation of the instance.

See the class description for details.

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.

4.120. API reference - Class Device

Notation used in Ruby API documentation

Module: db

Description: A device inside a circuit.

Class hierarchy: Device » NetlistObject

Device object represent atomic devices such as resistors, diodes or transistors. The Device class represents a particular device with specific parameters. The type of device is represented by a DeviceClass object. Device objects live in Circuit objects, the device class objects live in the Netlist object.

Devices connect to nets through terminals. Terminals are described by a terminal ID which is essentially the zero-based index of the terminal. Terminal definitions can be obtained from the device class using the DeviceClass#terminal_definitions method.

Devices connect to nets through the Device#connect_terminal method. Device terminals can be disconnected using Device#disconnect_terminal.

Device objects are created inside a circuit with Circuit#create_device.

This class has been added in version 0.26.

Public methods

	void	**_assign**	(const Device other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new Device ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	const Circuit ptr	**circuit**		Gets the circuit the device lives in.
	Circuit ptr	**circuit**		Gets the circuit the device lives in (non-const version).
	void	**connect_terminal**	(unsigned long terminal_id, Net ptr net)	Connects the given terminal to the specified net.
	void	**connect_terminal**	(string terminal_name, Net ptr net)	Connects the given terminal to the specified net.
[const]	const DeviceAbstract ptr	**device_abstract**		Gets the device abstract for this device instance.

[const]	const DeviceClass ptr	**device_class**		Gets the device class the device belongs to.
	void	**disconnect_terminal**	(unsigned long terminal_id)	Disconnects the given terminal from any net.
	void	**disconnect_terminal**	(string terminal_name)	Disconnects the given terminal from any net.
[const,iter]	DeviceAbstractRef	**each_combined_abstra**		Iterates over the combined device specifications.
[const,iter]	DeviceReconnectedTerminal	**each_reconnected_terminal_for**	(unsigned long terminal_id)	Iterates over the reconnected terminal specifications for a given outer terminal.
[const]	string	**expanded_name**		Gets the expanded name of the device.
[const]	unsigned long	**id**		Gets the device ID.
[const]	bool	**is_combined_device?**		Returns true, if the device is a combined device.
[const]	string	**name**		Gets the name of the device.
	void	**name=**	(string name)	Sets the name of the device.
[const]	const Net ptr	**net_for_terminal**	(unsigned long terminal_id)	Gets the net connected to the specified terminal.
	Net ptr	**net_for_terminal**	(unsigned long terminal_id)	Gets the net connected to the specified terminal (non-const version).
[const]	const Net ptr	**net_for_terminal**	(string terminal_name)	Gets the net connected to the specified terminal.
	Net ptr	**net_for_terminal**	(string terminal_name)	Gets the net connected to the specified terminal (non-const version).
[const]	double	**parameter**	(unsigned long param_id)	Gets the parameter value for the given parameter ID.
[const]	double	**parameter**	(string param_name)	Gets the parameter value for the given parameter name.
	void	**set_parameter**	(unsigned long param_id, double value)	Sets the parameter value for the given parameter ID.
	void	**set_parameter**	(string param_name, double value)	Sets the parameter value for the given parameter name.
[const]	DCplxTrans	**trans**		Gets the location of the device.
	void	**trans=**	(const DCplxTrans t)	Sets the location of the device.

Detailed description

_assign	**Signature**: void **_assign** (const Device other) **Description**: Assigns another object to self
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_dup	**Signature**: *[const]* new Device ptr **_dup** **Description**: Creates a copy of self
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.

circuit

(1) Signature: *[const]* const Circuit ptr **circuit**

Description: Gets the circuit the device lives in.

(2) Signature: Circuit ptr **circuit**

Description: Gets the circuit the device lives in (non-const version).

This constness variant has been introduced in version 0.26.8

connect_terminal

(1) Signature: void **connect_terminal** (unsigned long terminal_id, Net ptr net)

Description: Connects the given terminal to the specified net.

(2) Signature: void **connect_terminal** (string terminal_name, Net ptr net)

Description: Connects the given terminal to the specified net.

This version accepts a terminal name. If the name is not a valid terminal name, an exception is raised. If the terminal has been connected to a global net, it will be disconnected from there.

device_abstract

Signature: *[const]* const DeviceAbstract ptr **device_abstract**

Description: Gets the device abstract for this device instance.

See DeviceAbstract for more details.

Python specific notes:
The object exposes a readable attribute 'device_abstract'. This is the getter.

device_class

Signature: *[const]* const DeviceClass ptr **device_class**

Description: Gets the device class the device belongs to.

disconnect_terminal

(1) Signature: void **disconnect_terminal** (unsigned long terminal_id)

Description: Disconnects the given terminal from any net.

If the terminal has been connected to a global, this connection will be disconnected too.

(2) Signature: void **disconnect_terminal** (string terminal_name)

Description: Disconnects the given terminal from any net.

This version accepts a terminal name. If the name is not a valid terminal name, an exception is raised.

each_combined_abstract

Signature: *[const,iter]* DeviceAbstractRef **each_combined_abstract**

Description: Iterates over the combined device specifications.

This feature applies to combined devices. This iterator will deliver all device abstracts present in addition to the default device abstract.

each_reconnected_terminal_f

Signature: *[const,iter]* DeviceReconnectedTerminal **each_reconnected_terminal_for** (unsigned long terminal_id)

Description: Iterates over the reconnected terminal specifications for a given outer terminal.

This feature applies to combined devices. This iterator will deliver all device-to-abstract terminal reroutings.

expanded_name

Signature: *[const]* string **expanded_name**

Description: Gets the expanded name of the device.

The expanded name takes the name of the device. If the name is empty, the numeric ID will be used to build a name.

id

Signature: *[const]* unsigned long **id**

Description: Gets the device ID.

The ID is a unique integer which identifies the device. It can be used to retrieve the device from the circuit using Circuit#device_by_id. When assigned, the device ID is not 0.

is_combined_device?

Signature: *[const]* bool **is_combined_device?**

Description: Returns true, if the device is a combined device.

Combined devices feature multiple device abstracts and device-to-abstract terminal connections. See each_reconnected_terminal and each_combined_abstract for more details.

name

Signature: *[const]* string **name**

Description: Gets the name of the device.

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string name)

Description: Sets the name of the device.

Device names are used to name a device inside a netlist file. Device names should be unique within a circuit.

Python specific notes:
The object exposes a writable attribute 'name'. This is the setter.

net_for_terminal

(1) Signature: *[const]* const Net ptr **net_for_terminal** (unsigned long terminal_id)

Description: Gets the net connected to the specified terminal.

If the terminal is not connected, nil is returned for the net.

(2) Signature: Net ptr **net_for_terminal** (unsigned long terminal_id)

Description: Gets the net connected to the specified terminal (non-const version).

If the terminal is not connected, nil is returned for the net.

This constness variant has been introduced in version 0.26.8

(3) Signature: *[const]* const Net ptr **net_for_terminal** (string terminal_name)

Description: Gets the net connected to the specified terminal.

If the terminal is not connected, nil is returned for the net.

This convenience method has been introduced in version 0.27.3.

(4) Signature: Net ptr **net_for_terminal** (string terminal_name)

Description: Gets the net connected to the specified terminal (non-const version).

If the terminal is not connected, nil is returned for the net.

	This convenience method has been introduced in version 0.27.3.
parameter	**(1) Signature**: *[const]* double **parameter** (unsigned long param_id) **Description**: Gets the parameter value for the given parameter ID. **(2) Signature**: *[const]* double **parameter** (string param_name) **Description**: Gets the parameter value for the given parameter name. If the parameter name is not valid, an exception is thrown.
set_parameter	**(1) Signature**: void **set_parameter** (unsigned long param_id, double value) **Description**: Sets the parameter value for the given parameter ID. **(2) Signature**: void **set_parameter** (string param_name, double value) **Description**: Sets the parameter value for the given parameter name. If the parameter name is not valid, an exception is thrown.
trans	**Signature**: *[const]* DCplxTrans **trans** **Description**: Gets the location of the device. See trans= for details about this method. **Python specific notes:** The object exposes a readable attribute 'trans'. This is the getter.
trans=	**Signature**: void **trans=** (const DCplxTrans t) **Description**: Sets the location of the device. The device location is essentially describing the position of the device. The position is typically the center of some recognition shape. In this case the transformation is a plain displacement to the center of this shape. **Python specific notes:** The object exposes a writable attribute 'trans'. This is the setter.

4.121. API reference - Class DeviceAbstract

Notation used in Ruby API documentation

Module: db

Description: A geometrical device abstract

This class represents the geometrical model for the device. It links into the extracted layout to a cell which holds the terminal shapes for the device.

This class has been added in version 0.26.

Public constructors

new DeviceAbstract ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const DeviceAbstra other)	Assigns another object to self
[const]	unsigned int	**cell_index**		Gets the cell index of the device abstract.
[const]	unsigned long	**cluster_id_for_termina**	(unsigned long terminal_id)	Gets the cluster ID for the given terminal.
[const]	const DeviceClass ptr	**device_class**		Gets the device class of the device.
[const]	new DeviceAbstract ptr	**dup**		Creates a copy of self
[const]	string	**name**		Gets the name of the device abstract.
	void	**name=**	(string name)	Sets the name of the device abstract.
[const]	const Netlist ptr	**netlist**		Gets the netlist the device abstract lives in.
	Netlist ptr	**netlist**		Gets the netlist the device abstract lives in (non-const version).

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

	will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
assign	**Signature**: void **assign** (const DeviceAbstract other) **Description**: Assigns another object to self
cell_index	**Signature**: *[const]* unsigned int **cell_index** **Description**: Gets the cell index of the device abstract. This is the cell that represents the device.
cluster_id_for_terminal	**Signature**: *[const]* unsigned long **cluster_id_for_terminal** (unsigned long terminal_id) **Description**: Gets the cluster ID for the given terminal. The cluster ID links the terminal to geometrical shapes within the clusters of the cell (see cell_index)
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
device_class	**Signature**: *[const]* const DeviceClass ptr **device_class** **Description**: Gets the device class of the device.
dup	**Signature**: *[const]* new DeviceAbstract ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference

	Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
name	**Signature**: *[const]* string **name** **Description**: Gets the name of the device abstract. **Python specific notes:** The object exposes a readable attribute 'name'. This is the getter.
name=	**Signature**: void **name=** (string name) **Description**: Sets the name of the device abstract. Device names are used to name a device abstract inside a netlist file. Device names should be unique within a netlist. **Python specific notes:** The object exposes a writable attribute 'name'. This is the setter.
netlist	**(1) Signature**: *[const]* const Netlist ptr **netlist** **Description**: Gets the netlist the device abstract lives in. **(2) Signature**: Netlist ptr **netlist** **Description**: Gets the netlist the device abstract lives in (non-const version). This constness variant has been introduced in version 0.26.8
new	**Signature**: *[static]* new DeviceAbstract ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.

4.122. API reference - Class SubCircuit

Notation used in Ruby API documentation

Module: db

Description: A subcircuit inside a circuit.

Class hierarchy: SubCircuit » NetlistObject

Circuits may instantiate other circuits as subcircuits similar to cells in layouts. Such an instance is a subcircuit. A subcircuit refers to a circuit implementation (a Circuit object), and presents connections through pins. The pins of a subcircuit can be connected to nets. The subcircuit pins are identical to the outgoing pins of the circuit the subcircuit refers to.

Subcircuits connect to nets through the SubCircuit#connect_pin method. SubCircuit pins can be disconnected using SubCircuit#disconnect_pin.

Subcircuit objects are created inside a circuit with Circuit#create_subcircuit.

This class has been added in version 0.26.

Public methods

	void	**_assign**	(const SubCircuit other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new SubCircuit ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	const Circuit ptr	**circuit**		Gets the circuit the subcircuit lives in.
	Circuit ptr	**circuit**		Gets the circuit the subcircuit lives in (non-const version).
[const]	const Circuit ptr	**circuit_ref**		Gets the circuit referenced by the subcircuit.
	Circuit ptr	**circuit_ref**		Gets the circuit referenced by the subcircuit (non-const version).
	void	**connect_pin**	(unsigned long pin_id, Net ptr net)	Connects the given pin to the specified net.
	void	**connect_pin**	(const Pin ptr pin, Net ptr net)	Connects the given pin to the specified net.

	void	**disconnect_pin**	(unsigned long pin_id)	Disconnects the given pin from any net.
	void	**disconnect_pin**	(const Pin ptr pin)	Disconnects the given pin from any net.
[const]	string	**expanded_name**		Gets the expanded name of the subcircuit.
[const]	unsigned long	**id**		Gets the subcircuit ID.
[const]	string	**name**		Gets the name of the subcircuit.
	void	**name=**	(string name)	Sets the name of the subcircuit.
[const]	const Net ptr	**net_for_pin**	(unsigned long pin_id)	Gets the net connected to the specified pin of the subcircuit.
	Net ptr	**net_for_pin**	(unsigned long pin_id)	Gets the net connected to the specified pin of the subcircuit (non-const version).
[const]	DCplxTrans	**trans**		Gets the physical transformation for the subcircuit.
	void	**trans=**	(const DCplxTrans trans)	Sets the physical transformation for the subcircuit.

Detailed description

_assign	**Signature**: void **_assign** (const SubCircuit other) **Description**: Assigns another object to self
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_dup	**Signature**: *[const]* new SubCircuit ptr **_dup** **Description**: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

circuit

(1) Signature: *[const]* const Circuit ptr **circuit**

Description: Gets the circuit the subcircuit lives in.

This is NOT the circuit which is referenced. For getting the circuit that the subcircuit references, use circuit_ref.

(2) Signature: Circuit ptr **circuit**

Description: Gets the circuit the subcircuit lives in (non-const version).

This is NOT the circuit which is referenced. For getting the circuit that the subcircuit references, use circuit_ref.

This constness variant has been introduced in version 0.26.8

circuit_ref

(1) Signature: *[const]* const Circuit ptr **circuit_ref**

Description: Gets the circuit referenced by the subcircuit.

(2) Signature: Circuit ptr **circuit_ref**

Description: Gets the circuit referenced by the subcircuit (non-const version).

This constness variant has been introduced in version 0.26.8

connect_pin

(1) Signature: void **connect_pin** (unsigned long pin_id, Net ptr net)

Description: Connects the given pin to the specified net.

(2) Signature: void **connect_pin** (const Pin ptr pin, Net ptr net)

Description: Connects the given pin to the specified net.

	This version takes a Pin reference instead of a pin ID.
disconnect_pin	**(1) Signature**: void **disconnect_pin** (unsigned long pin_id) **Description**: Disconnects the given pin from any net. **(2) Signature**: void **disconnect_pin** (const Pin ptr pin) **Description**: Disconnects the given pin from any net. This version takes a Pin reference instead of a pin ID.
expanded_name	**Signature**: *[const]* string **expanded_name** **Description**: Gets the expanded name of the subcircuit. The expanded name takes the name of the subcircuit. If the name is empty, the numeric ID will be used to build a name.
id	**Signature**: *[const]* unsigned long **id** **Description**: Gets the subcircuit ID. The ID is a unique integer which identifies the subcircuit. It can be used to retrieve the subcircuit from the circuit using Circuit#subcircuit_by_id. When assigned, the subcircuit ID is not 0.
name	**Signature**: *[const]* string **name** **Description**: Gets the name of the subcircuit. **Python specific notes:** The object exposes a readable attribute 'name'. This is the getter.
name=	**Signature**: void **name=** (string name) **Description**: Sets the name of the subcircuit. SubCircuit names are used to name a subcircuits inside a netlist file. SubCircuit names should be unique within a circuit. **Python specific notes:** The object exposes a writable attribute 'name'. This is the setter.
net_for_pin	**(1) Signature**: *[const]* const Net ptr **net_for_pin** (unsigned long pin_id) **Description**: Gets the net connected to the specified pin of the subcircuit. If the pin is not connected, nil is returned for the net. **(2) Signature**: Net ptr **net_for_pin** (unsigned long pin_id) **Description**: Gets the net connected to the specified pin of the subcircuit (non-const version). If the pin is not connected, nil is returned for the net. This constness variant has been introduced in version 0.26.8
trans	**Signature**: *[const]* DCplxTrans **trans** **Description**: Gets the physical transformation for the subcircuit. This property applies to subcircuits derived from a layout. It specifies the placement of the respective cell. This property has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'trans'. This is the getter.

trans=

Signature: void **trans=** (const DCplxTrans trans)

Description: Sets the physical transformation for the subcircuit.

See trans for details about this property.

This property has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.

4.123. API reference - Class NetTerminalRef

Notation used in Ruby API documentation

Module: db

Description: A connection to a terminal of a device.

This object is used inside a net (see Net) to describe the connections a net makes.

This class has been added in version 0.26.

Public constructors

new NetTerminalRef ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetTerm other)	Assigns another object to self
[const]	const Device ptr	**device**		Gets the device reference.
	Device ptr	**device**		Gets the device reference (non-const version).
[const]	const DeviceClass ptr	**device_class**		Gets the class of the device which is addressed.
[const]	new NetTerminalRef ptr	**dup**		Creates a copy of self
[const]	const Net ptr	**net**		Gets the net this terminal reference is attached to.
	Net ptr	**net**		Gets the net this terminal reference is attached to (non-const version).
[const]	const DeviceTerminalDefinition ptr	**terminal_def**		Gets the terminal definition of the terminal that is connected
[const]	unsigned long	**terminal_id**		Gets the ID of the terminal of the device the connection is made to.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const NetTerminalRef other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

device

(1) Signature: *[const]* const Device ptr **device**

Description: Gets the device reference.

Gets the device object that this connection is made to.

(2) Signature: Device ptr **device**

Description: Gets the device reference (non-const version).

Gets the device object that this connection is made to.

This constness variant has been introduced in version 0.26.8

device_class

Signature: *[const]* const DeviceClass ptr **device_class**

Description: Gets the class of the device which is addressed.

dup

Signature: *[const]* new NetTerminalRef ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
net	**(1) Signature**: *[const]* const Net ptr **net** **Description**: Gets the net this terminal reference is attached to. **(2) Signature**: Net ptr **net** **Description**: Gets the net this terminal reference is attached to (non-const version). This constness variant has been introduced in version 0.26.8
new	**Signature**: *[static]* new NetTerminalRef ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
terminal_def	**Signature**: *[const]* const DeviceTerminalDefinition ptr **terminal_def** **Description**: Gets the terminal definition of the terminal that is connected
terminal_id	**Signature**: *[const]* unsigned long **terminal_id** **Description**: Gets the ID of the terminal of the device the connection is made to.

4.124. API reference - Class NetPinRef

Notation used in Ruby API documentation

Module: db

Description: A connection to an outgoing pin of the circuit.

This object is used inside a net (see Net) to describe the connections a net makes.

This class has been added in version 0.26.

Public constructors

new NetPinRef ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetPinR other)	Assigns another object to self
[const]	new NetPinRef ptr	**dup**		Creates a copy of self
[const]	const Net ptr	**net**		Gets the net this pin reference is attached to.
	Net ptr	**net**		Gets the net this pin reference is attached to (non-const version).
[const]	const Pin ptr	**pin**		Gets the Pin object of the pin the connection is made to.
[const]	unsigned long	**pin_id**		Gets the ID of the pin the connection is made to.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead

[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const NetPinRef other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetPinRef ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

net

(1) Signature: *[const]* const Net ptr **net**

Description: Gets the net this pin reference is attached to.

(2) Signature: Net ptr **net**

Description: Gets the net this pin reference is attached to (non-const version).

This constness variant has been introduced in version 0.26.8

new

Signature: *[static]* new NetPinRef ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

pin

Signature: *[const]* const Pin ptr **pin**

Description: Gets the Pin object of the pin the connection is made to.

pin_id

Signature: *[const]* unsigned long **pin_id**

Description: Gets the ID of the pin the connection is made to.

4.125. API reference - Class NetSubcircuitPinRef

Notation used in Ruby API documentation

Module: db

Description: A connection to a pin of a subcircuit.

This object is used inside a net (see Net) to describe the connections a net makes.

This class has been added in version 0.26.

Public constructors

new NetSubcircuitPinRef ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetSubci other)	Assigns another object to self
[const]	new NetSubcircuitPinRef ptr	**dup**		Creates a copy of self
[const]	const Net ptr	**net**		Gets the net this pin reference is attached to.
	Net ptr	**net**		Gets the net this pin reference is attached to (non-const version).
[const]	const Pin ptr	**pin**		Gets the Pin object of the pin the connection is made to.
[const]	unsigned long	**pin_id**		Gets the ID of the pin the connection is made to.
[const]	const SubCircuit ptr	**subcircuit**		Gets the subcircuit reference.
	SubCircuit ptr	**subcircuit**		Gets the subcircuit reference (non-const version).

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead

	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const NetSubcircuitPinRef other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetSubcircuitPinRef ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

net

(1) Signature: *[const]* const Net ptr **net**

Description: Gets the net this pin reference is attached to.

(2) Signature: Net ptr **net**

Description: Gets the net this pin reference is attached to (non-const version).

This constness variant has been introduced in version 0.26.8

new

Signature: *[static]* new NetSubcircuitPinRef ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

pin

Signature: *[const]* const Pin ptr **pin**

Description: Gets the Pin object of the pin the connection is made to.

pin_id

Signature: *[const]* unsigned long **pin_id**

Description: Gets the ID of the pin the connection is made to.

subcircuit

(1) Signature: *[const]* const SubCircuit ptr **subcircuit**

Description: Gets the subcircuit reference.

This attribute indicates the subcircuit the net attaches to. The subcircuit lives in the same circuit than the net.

(2) Signature: SubCircuit ptr **subcircuit**

Description: Gets the subcircuit reference (non-const version).

This attribute indicates the subcircuit the net attaches to. The subcircuit lives in the same circuit than the net.

This constness variant has been introduced in version 0.26.8

4.126. API reference - Class Net

Notation used in Ruby API documentation

Module: db

Description: A single net.

Class hierarchy: Net » NetlistObject

A net connects multiple pins or terminals together. Pins are either pin or subcircuits of outgoing pins of the circuit the net lives in. Terminals are connections made to specific terminals of devices.

Net objects are created inside a circuit with Circuit#create_net.

To connect a net to an outgoing pin of a circuit, use Circuit#connect_pin, to disconnect a net from an outgoing pin use Circuit#disconnect_pin. To connect a net to a pin of a subcircuit, use SubCircuit#connect_pin, to disconnect a net from a pin of a subcircuit, use SubCircuit#disconnect_pin. To connect a net to a terminal of a device, use Device#connect_terminal, to disconnect a net from a terminal of a device, use Device#disconnect_terminal.

This class has been added in version 0.26.

Public methods

	void	**_assign**	(const Net other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new Net ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	Circuit ptr	**circuit**		Gets the circuit the net lives in.
	void	**clear**		Clears the net.
[const]	unsigned long	**cluster_id**		Gets the cluster ID of the net.
	void	**cluster_id=**	(unsigned long id)	Sets the cluster ID of the net.
[const,iter]	NetPinRef	**each_pin**		Iterates over all outgoing pins the net connects.
[iter]	NetPinRef	**each_pin**		Iterates over all outgoing pins the net connects (non-const version).
[const,iter]	NetSubcircuitPinRef	**each_subcircuit_pin**		Iterates over all subcircuit pins the net connects.

[iter]	NetSubcircuitPinRef	**each_subcircuit_pin**		Iterates over all subcircuit pins the net connects (non-const version).
[const,iter]	NetTerminalRef	**each_terminal**		Iterates over all terminals the net connects.
[iter]	NetTerminalRef	**each_terminal**		Iterates over all terminals the net connects (non-const version).
[const]	string	**expanded_name**		Gets the expanded name of the net.
[const]	bool	**is_floating?**		Returns true, if the net is floating.
[const]	bool	**is_internal?**		Returns true, if the net is an internal net.
[const]	bool	**is_passive?**		Returns true, if the net is passive.
[const]	string	**name**		Gets the name of the net.
	void	**name=**	(string name)	Sets the name of the net.
[const]	unsigned long	**pin_count**		Returns the number of outgoing pins connected by this net.
[const]	string	**qname**		Gets the qualified name.
[const]	unsigned long	**subcircuit_pin_count**		Returns the number of subcircuit pins connected by this net.
[const]	unsigned long	**terminal_count**		Returns the number of terminals connected by this net.
[const]	string	**to_s**		Gets the qualified name.

Detailed description

_assign

Signature: void **_assign** (const Net other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new Net ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

circuit

Signature: Circuit ptr **circuit**

Description: Gets the circuit the net lives in.

clear

Signature: void **clear**

Description: Clears the net.

cluster_id

Signature: *[const]* unsigned long **cluster_id**

Description: Gets the cluster ID of the net.

See cluster_id= for details about the cluster ID.

Python specific notes:
The object exposes a readable attribute 'cluster_id'. This is the getter.

cluster_id=

Signature: void **cluster_id=** (unsigned long id)

Description: Sets the cluster ID of the net.

The cluster ID connects the net with a layout cluster. It is set when the net is extracted from a layout.

Python specific notes:
The object exposes a writable attribute 'cluster_id'. This is the setter.

each_pin

(1) Signature: *[const,iter]* NetPinRef **each_pin**

Description: Iterates over all outgoing pins the net connects.

Pin connections are described by NetPinRef objects. Pin connections are connections to outgoing pins of the circuit the net lives in.

(2) Signature: *[iter]* NetPinRef **each_pin**

Description: Iterates over all outgoing pins the net connects (non-const version).

Pin connections are described by NetPinRef objects. Pin connections are connections to outgoing pins of the circuit the net lives in.

This constness variant has been introduced in version 0.26.8

each_subcircuit_pin

(1) Signature: *[const,iter]* NetSubcircuitPinRef **each_subcircuit_pin**

Description: Iterates over all subcircuit pins the net connects.

Subcircuit pin connections are described by NetSubcircuitPinRef objects. These are connections to specific pins of subcircuits.

(2) Signature: *[iter]* NetSubcircuitPinRef **each_subcircuit_pin**

Description: Iterates over all subcircuit pins the net connects (non-const version).

Subcircuit pin connections are described by NetSubcircuitPinRef objects. These are connections to specific pins of subcircuits.

This constness variant has been introduced in version 0.26.8

each_terminal

(1) Signature: *[const,iter]* NetTerminalRef **each_terminal**

Description: Iterates over all terminals the net connects.

Terminals connect devices. Terminal connections are described by NetTerminalRef objects.

(2) Signature: *[iter]* NetTerminalRef **each_terminal**

Description: Iterates over all terminals the net connects (non-const version).

Terminals connect devices. Terminal connections are described by NetTerminalRef objects.

This constness variant has been introduced in version 0.26.8

expanded_name

Signature: *[const]* string **expanded_name**

Description: Gets the expanded name of the net.

The expanded name takes the name of the net. If the name is empty, the cluster ID will be used to build a name.

is_floating?

Signature: *[const]* bool **is_floating?**

Description: Returns true, if the net is floating.

Floating nets are those which don't have any device or subcircuit on it and are not connected through a pin.

is_internal?

Signature: *[const]* bool **is_internal?**

Description: Returns true, if the net is an internal net.

Internal nets are those which connect exactly two terminals and nothing else (pin_count = 0 and terminal_count == 2).

is_passive?

Signature: *[const]* bool **is_passive?**

Description: Returns true, if the net is passive.

Passive nets don't have devices or subcircuits on it. They can be exposed through a pin. is_floating? implies is_passive?.

This method has been introduced in version 0.26.1.

name

Signature: *[const]* string **name**

Description: Gets the name of the net.

See name= for details about the name.

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string name)

Description: Sets the name of the net.

The name of the net is used for naming the net in schematic files for example. The name of the net has to be unique.

Python specific notes:
The object exposes a writable attribute 'name'. This is the setter.

pin_count

Signature: *[const]* unsigned long **pin_count**

Description: Returns the number of outgoing pins connected by this net.

qname

Signature: *[const]* string **qname**

Description: Gets the qualified name.

The qualified name is like the expanded name, but the circuit's name is preceded (i.e. 'CIRCUIT:NET') if available.

Python specific notes:
This method is also available as 'str(object)'.

subcircuit_pin_count

Signature: *[const]* unsigned long **subcircuit_pin_count**

Description: Returns the number of subcircuit pins connected by this net.

terminal_count

Signature: *[const]* unsigned long **terminal_count**

Description: Returns the number of terminals connected by this net.

to_s

Signature: *[const]* string **to_s**

Description: Gets the qualified name.

The qualified name is like the expanded name, but the circuit's name is preceded (i.e. 'CIRCUIT:NET') if available.

Python specific notes:
This method is also available as 'str(object)'.

4.127. API reference - Class DeviceTerminalDefinition

Notation used in Ruby API documentation

Module: db

Description: A terminal descriptor

This class is used inside the DeviceClass class to describe a terminal of the device.

This class has been added in version 0.26.

Public constructors

new DeviceTerminalDefinition ptr	**new**	(string name, string description =)	Creates a new terminal definition.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const DeviceTermin other)	Assigns another object to self
[const]	string	**description**		Gets the description of the terminal.
	void	**description=**	(string description)	Sets the description of the terminal.
[const]	new DeviceTerminalDefinition ptr	**dup**		Creates a copy of self
[const]	unsigned long	**id**		Gets the ID of the terminal.
[const]	string	**name**		Gets the name of the terminal.
	void	**name=**	(string name)	Sets the name of the terminal.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead

	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const DeviceTerminalDefinition other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

description

Signature: *[const]* string **description**

Description: Gets the description of the terminal.

Python specific notes:
The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string description)

Description: Sets the description of the terminal.

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new DeviceTerminalDefinition ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

id

Signature: *[const]* unsigned long **id**

Description: Gets the ID of the terminal.

The ID of the terminal is used in some places to refer to a specific terminal (e.g. in the NetTerminalRef object).

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

	This method returns true, if self is a const reference. In that case, only const methods may be called on self.
name	**Signature**: *[const]* string **name** **Description**: Gets the name of the terminal. **Python specific notes:** The object exposes a readable attribute 'name'. This is the getter.
name=	**Signature**: void **name=** (string name) **Description**: Sets the name of the terminal. **Python specific notes:** The object exposes a writable attribute 'name'. This is the setter.
new	**Signature**: *[static]* new DeviceTerminalDefinition ptr **new** (string name, string description =) **Description**: Creates a new terminal definition. **Python specific notes:** This method is the default initializer of the object.

4.128. API reference - Class DeviceParameterDefinition

Notation used in Ruby API documentation

Module: db

Description: A parameter descriptor

This class is used inside the DeviceClass class to describe a parameter of the device.

This class has been added in version 0.26.

Public constructors

new DeviceParameterDefinition ptr	**new**	(string name, string description = , double default_value = 0, bool is_primary = true, double si_scaling = 1, double geo_scaling_exponent = 0)	Creates a new parameter definition.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const DeviceParamet other)	Assigns another object to self
[const]	double	**default_value**		Gets the default value of the parameter.
	void	**default_value=**	(double default_value)	Sets the default value of the parameter.
[const]	string	**description**		Gets the description of the parameter.
	void	**description=**	(string description)	Sets the description of the parameter.
[const]	new DeviceParameterDefinition ptr	**dup**		Creates a copy of self
[const]	double	**geo_scaling_exponent**		Gets the geometry scaling exponent.

	void	geo_scaling_exponent=	(double expo)	Sets the geometry scaling exponent.
[const]	unsigned long	id		Gets the ID of the parameter.
	void	is_primary=	(bool primary)	Sets a value indicating whether the parameter is a primary parameter
[const]	bool	is_primary?		Gets a value indicating whether the parameter is a primary parameter
[const]	string	name		Gets the name of the parameter.
	void	name=	(string name)	Sets the name of the parameter.
[const]	double	si_scaling		Gets the scaling factor to SI units.
	void	si_scaling=	(double flag)	Sets the scaling factor to SI units.

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const DeviceParameterDefinition other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

default_value

Signature: *[const]* double **default_value**

Description: Gets the default value of the parameter.

Python specific notes:
The object exposes a readable attribute 'default_value'. This is the getter.

default_value=

Signature: void **default_value=** (double default_value)

Description: Sets the default value of the parameter.

The default value is used to initialize parameters of Device objects.

Python specific notes:
The object exposes a writable attribute 'default_value'. This is the setter.

description

Signature: *[const]* string **description**

Description: Gets the description of the parameter.

Python specific notes:

The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string description)

Description: Sets the description of the parameter.

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new DeviceParameterDefinition ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

geo_scaling_exponent

Signature: *[const]* double **geo_scaling_exponent**

Description: Gets the geometry scaling exponent.

This value is used when applying '.options scale' in the SPICE reader for example. It is zero for 'no scaling', 1.0 for linear scaling and 2.0 for quadratic scaling.

This attribute has been added in version 0.28.6.

Python specific notes:
The object exposes a readable attribute 'geo_scaling_exponent'. This is the getter.

geo_scaling_exponent=

Signature: void **geo_scaling_exponent=** (double expo)

Description: Sets the geometry scaling exponent.

See geo_scaling_exponent for details.

This attribute has been added in version 0.28.6.

Python specific notes:
The object exposes a writable attribute 'geo_scaling_exponent'. This is the setter.

id

Signature: *[const]* unsigned long **id**

Description: Gets the ID of the parameter.

The ID of the parameter is used in some places to refer to a specific parameter (e.g. in the NetParameterRef object).

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_primary=

Signature: void **is_primary=** (bool primary)

Description: Sets a value indicating whether the parameter is a primary parameter

If this flag is set to true (the default), the parameter is considered a primary parameter. Only primary parameters are compared by default.

Python specific notes:
The object exposes a writable attribute 'is_primary'. This is the setter.

is_primary?

Signature: *[const]* bool **is_primary?**

Description: Gets a value indicating whether the parameter is a primary parameter

See is_primary= for details about this predicate.

Python specific notes:
The object exposes a readable attribute 'is_primary'. This is the getter.

name

Signature: *[const]* string **name**

Description: Gets the name of the parameter.

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string name)

Description: Sets the name of the parameter.

Python specific notes:
The object exposes a writable attribute 'name'. This is the setter.

new

Signature: *[static]* new DeviceParameterDefinition ptr **new** (string name, string description = , double default_value = 0, bool is_primary = true, double si_scaling = 1, double geo_scaling_exponent = 0)

Description: Creates a new parameter definition.

name:	The name of the parameter
description:	The human-readable description
default_value:	The initial value
is_primary:	True, if the parameter is a primary parameter (see is_primary=)
si_scaling:	The scaling factor to SI units
geo_scaling_exponent:	Indicates how the parameter scales with geometrical scaling (0: no scaling, 1.0: linear, 2.0: quadratic)

Python specific notes:
This method is the default initializer of the object.

si_scaling

Signature: *[const]* double **si_scaling**

Description: Gets the scaling factor to SI units.

For parameters in micrometers - for example W and L of MOS devices - this factor can be set to 1e-6 to reflect the unit.

Python specific notes:
The object exposes a readable attribute 'si_scaling'. This is the getter.

si_scaling=

Signature: void **si_scaling=** (double flag)

Description: Sets the scaling factor to SI units.

This setter has been added in version 0.28.6.

Python specific notes:
The object exposes a writable attribute 'si_scaling'. This is the setter.

4.129. API reference - Class EqualDeviceParameters

Notation used in Ruby API documentation

Module: db

Description: A device parameter equality comparer.

Attach this object to a device class with DeviceClass#equal_parameters= to make the device class use this comparer:

```
# 20nm tolerance for length:
equal_device_parameters = RBA::EqualDeviceParameters::new(RBA::DeviceClassMOS4Transistor::PARAM_L, 0.02, 0.0)
# one percent tolerance for width:
equal_device_parameters += RBA::EqualDeviceParameters::new(RBA::DeviceClassMOS4Transistor::PARAM_W, 0.0,
 0.01)
# applies the compare delegate:
netlist.device_class_by_name("NMOS").equal_parameters = equal_device_parameters
```

You can use this class to specify fuzzy equality criteria for the comparison of device parameters in netlist verification or to confine the equality of devices to certain parameters only.

This class has been added in version 0.26.

Public constructors

new EqualDeviceParameters ptr	**new**	(unsigned long param_id, double absolute = 0, double relative = 0)	Creates a device parameter comparer for a single parameter.

Public methods

[const]	EqualDeviceParameters	**+**	(const EqualDevicePara other)	Combines two parameters for comparison.
[const]	EqualDeviceParameters	**+=**	(const EqualDeviceParameters other)	Combines two parameters for comparison (in-place).
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const EqualDevicePara other)	Assigns another object to self

[const]	new EqualDeviceParameters ptr	**dup**	Creates a copy of self

Public static methods and constants

	new EqualDeviceParameters ptr	**ignore**	(unsigned long param_id)	Creates a device parameter comparer which ignores the parameter.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

+

Signature: *[const]* EqualDeviceParameters **+** (const EqualDeviceParameters other)

Description: Combines two parameters for comparison.

The '+' operator will join the parameter comparers and produce one that checks the combined parameters.

+=

Signature: *[const]* EqualDeviceParameters **+=** (const EqualDeviceParameters other)

Description: Combines two parameters for comparison (in-place).

The '+=' operator will join the parameter comparers and produce one that checks the combined parameters.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const EqualDeviceParameters other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new EqualDeviceParameters ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

ignore

Signature: *[static]* new EqualDeviceParameters ptr **ignore** (unsigned long param_id)

Description: Creates a device parameter comparer which ignores the parameter.

This specification can be used to make a parameter ignored. Starting with version 0.27.4, all primary parameters are compared. Before 0.27.4, giving a tolerance meant only those parameters are compared. To exclude a primary parameter from the compare, use the 'ignore' specification for that parameter.

This constructor has been introduced in version 0.27.4.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new EqualDeviceParameters ptr **new** (unsigned long param_id, double absolute = 0, double relative = 0)

Description: Creates a device parameter comparer for a single parameter.

'absolute' is the absolute deviation allowed for the parameter values. 'relative' is the relative deviation allowed for the parameter values (a value between 0 and 1).

A value of 0 for both absolute and relative deviation means the parameters have to match exactly.

If 'absolute' and 'relative' are both given, their deviations will add to the allowed difference between two parameter values. The relative deviation will be applied to the mean value of both parameter values. For example, when comparing parameter values of 40 and 60, a relative deviation of 0.35 means an absolute deviation of 17.5 (= 0.35 * average of 40 and 60) which does not make both values match.

Python specific notes:
This method is the default initializer of the object.

4.130. API reference - Class GenericDeviceParameterCompare

Notation used in Ruby API documentation

Module: db

Description: A class implementing the comparison of device parameters.

Class hierarchy: GenericDeviceParameterCompare » EqualDeviceParameters

Reimplement this class to provide a custom device parameter compare scheme. Attach this object to a device class with DeviceClass#equal_parameters= to make the device class use this comparer.

This class is intended for special cases. In most scenarios it is easier to use EqualDeviceParameters instead of implementing a custom comparer class.

This class has been added in version 0.26. The 'equal' method has been dropped in 0.27.1 as it can be expressed as !less(a,b) && ! less(b,a).

Public methods

	void	**_assign**	(const GenericDevice other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new GenericDeviceParame ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[virtual,const	bool	**less**	(const Device device_a, const Device device_b)	Compares the parameters of two devices for a begin less than b. Returns true, if the parameters of device a are considered less than those of device b.The 'less' implementation needs to ensure strict weak ordering. Specifically, less(a,b) == false and less(b,a) implies that a is equal to b and less(a,b) == true implies that less(b,a) is false and vice versa. If not, an internal error will be encountered on netlist compare.

Detailed description

_assign

Signature: void **_assign** (const GenericDeviceParameterCompare other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new GenericDeviceParameterCompare ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

less

Signature: *[virtual,const]* bool **less** (const Device device_a, const Device device_b)

Description: Compares the parameters of two devices for a begin less than b. Returns true, if the parameters of device a are considered less than those of device b.The 'less' implementation needs to ensure strict weak ordering. Specifically, less(a,b) == false and less(b,a) implies that a is equal to

b and less(a,b) == true implies that less(b,a) is false and vice versa. If not, an internal error will be encountered on netlist compare.

4.131. API reference - Class GenericDeviceCombiner

Notation used in Ruby API documentation

Module: db

Description: A class implementing the combination of two devices (parallel or serial mode).

Reimplement this class to provide a custom device combiner. Device combination requires 'supports_paralell_combination' or 'supports_serial_combination' to be set to true for the device class. In the netlist device combination step, the algorithm will try to identify devices which can be combined into single devices and use the combiner object to implement the actual joining of such devices.

Attach this object to a device class with DeviceClass#combiner= to make the device class use this combiner.

This class has been added in version 0.27.3.

Public constructors

new GenericDeviceCombiner ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const GenericDevice(other)	Assigns another object to self
[virtual,const]	bool	**combine_devices**	(Device ptr device_a, Device ptr device_b)	Combines two devices if possible.
[const]	new GenericDeviceCombine ptr	**dup**		Creates a copy of self

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead

[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const GenericDeviceCombiner other)

Description: Assigns another object to self

combine_devices

Signature: *[virtual,const]* bool **combine_devices** (Device ptr device_a, Device ptr device_b)

Description: Combines two devices if possible.

This method needs to test, whether the two devices can be combined. Both devices are guaranteed to share the same device class. If they cannot be combined, this method shall do nothing and return false. If they can be combined, this method shall reconnect the nets of the first device and entirely disconnect the nets of the second device. The second device will be deleted afterwards.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new GenericDeviceCombiner ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new GenericDeviceCombiner ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

4.132. API reference - Class DeviceClass

Notation used in Ruby API documentation

Module: db

Description: A class describing a specific type of device.

Device class objects live in the context of a Netlist object. After a device class is created, it must be added to the netlist using Netlist#add. The netlist will own the device class object. When the netlist is destroyed, the device class object will become invalid.

The DeviceClass class is the base class for other device classes.

This class has been added in version 0.26. In version 0.27.3, the 'GenericDeviceClass' has been integrated with DeviceClass and the device class was made writeable in most respects. This enables manipulating built-in device classes.

Public constructors

new DeviceClass ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add_parameter**	(DeviceParameterD ptr parameter_def)	Adds the given parameter definition to the device class
	void	**add_terminal**	(DeviceTerminalDefinition ptr terminal_def)	Adds the given terminal definition to the device class
	void	**assign**	(const DeviceClass other)	Assigns another object to self
	void	**clear_equivalent_terminal_ids**		Clears all equivalent terminal ids
	void	**clear_parameters**		Clears the list of parameters
	void	**clear_terminals**		Clears the list of terminals
	GenericDeviceCombine ptr	**combiner**		Gets a device combiner or nil if none is registered.

	void	**combiner=**	(GenericDeviceCombiner ptr combiner)	Specifies a device combiner (parallel or serial device combination).
[const]	string	**description**		Gets the description text of the device class.
	void	**description=**	(string description)	Sets the description of the device class.
[const]	new DeviceClass ptr	**dup**		Creates a copy of self
	void	**enable_parameter**	(unsigned long parameter_id, bool enable)	Enables or disables a parameter.
	void	**enable_parameter**	(string parameter_name, bool enable)	Enables or disables a parameter.
	EqualDeviceParameters ptr	**equal_parameters**		Gets the device parameter comparer for netlist verification or nil if no comparer is registered.
	void	**equal_parameters=**	(EqualDeviceParam ptr comparer)	Specifies a device parameter comparer for netlist verification.
	void	**equivalent_terminal_id**	(unsigned long original_id, unsigned long equivalent_id)	Specifies a terminal to be equivalent to another.
[const]	bool	**has_parameter?**	(string name)	Returns true, if the device class has a parameter with the given name.
[const]	bool	**has_terminal?**	(string name)	Returns true, if the device class has a terminal with the given name.
[const]	unsigned long	**id**		Gets the unique ID of the device class
[const]	string	**name**		Gets the name of the device class.
	void	**name=**	(string name)	Sets the name of the device class.
	Netlist ptr	**netlist**		Gets the netlist the device class lives in.
[const]	const DeviceParameterDefini ptr	**parameter_definition**	(unsigned long parameter_id)	Gets the parameter definition object for a given ID.
[const]	const DeviceParameterDefinition ptr	**parameter_definition**	(string parameter_name)	Gets the parameter definition object for a given ID.
[const]	DeviceParameterDefini	**parameter_definitions**		Gets the list of parameter definitions of the device.
[const]	unsigned long	**parameter_id**	(string name)	Returns the parameter ID of the parameter with the given name.

	void	strict=	(bool s)	Sets a value indicating whether this class performs strict terminal mapping
[const]	bool	strict?		Gets a value indicating whether this class performs strict terminal mapping
	void	supports_parallel_comb	(bool f)	Specifies whether the device supports parallel device combination.
	void	supports_serial_combinations=	(bool f)	Specifies whether the device supports serial device combination.
[const]	const DeviceTerminalDefiniti ptr	terminal_definition	(unsigned long terminal_id)	Gets the terminal definition object for a given ID.
[const]	DeviceTerminalDefinition[]	terminal_definitions		Gets the list of terminal definitions of the device.
[const]	unsigned long	terminal_id	(string name)	Returns the terminal ID of the terminal with the given name.

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_parameter

Signature: void **add_parameter** (DeviceParameterDefinition ptr parameter_def)

Description: Adds the given parameter definition to the device class

This method will define a new parameter. The new parameter is added at the end of existing parameters. The parameter definition object passed as the argument is modified to contain the new ID of the parameter. The parameter is copied into the device class. Modifying the parameter object later does not have the effect of changing the parameter definition.

This method has been moved from 'GenericDeviceClass' to 'DeviceClass' in version 0.27.3.

add_terminal

Signature: void **add_terminal** (DeviceTerminalDefinition ptr terminal_def)

Description: Adds the given terminal definition to the device class

This method will define a new terminal. The new terminal is added at the end of existing terminals. The terminal definition object passed as the argument is modified to contain the new ID of the terminal.

The terminal is copied into the device class. Modifying the terminal object later does not have the effect of changing the terminal definition.

This method has been moved from 'GenericDeviceClass' to 'DeviceClass' in version 0.27.3.

assign

Signature: void **assign** (const DeviceClass other)

Description: Assigns another object to self

clear_equivalent_terminal_ids

Signature: void **clear_equivalent_terminal_ids**

Description: Clears all equivalent terminal ids

This method has been added in version 0.27.3.

clear_parameters

Signature: void **clear_parameters**

Description: Clears the list of parameters

This method has been added in version 0.27.3.

clear_terminals

Signature: void **clear_terminals**

Description: Clears the list of terminals

This method has been moved from 'GenericDeviceClass' to 'DeviceClass' in version 0.27.3.

combiner

Signature: GenericDeviceCombiner ptr **combiner**

Description: Gets a device combiner or nil if none is registered.

This method has been added in version 0.27.3.

Python specific notes:
The object exposes a readable attribute 'combiner'. This is the getter.

combiner=

Signature: void **combiner=** (GenericDeviceCombiner ptr combiner)

Description: Specifies a device combiner (parallel or serial device combination).

You can assign nil for the combiner to remove it.

In special cases, you can even implement a custom combiner by deriving your own comparer from the GenericDeviceCombiner class.

This method has been added in version 0.27.3.

Python specific notes:
The object exposes a writable attribute 'combiner'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

description

Signature: *[const]* string **description**

Description: Gets the description text of the device class.

Python specific notes:
The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string description)

Description: Sets the description of the device class.

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new DeviceClass ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

enable_parameter

(1) Signature: void **enable_parameter** (unsigned long parameter_id, bool enable)

Description: Enables or disables a parameter.

Some parameters are 'secondary' parameters which are extracted but not handled in device compare and are not shown in the netlist browser. For example, the 'W' parameter of the resistor is such a secondary parameter. This method allows turning a parameter in a primary one ('enable') or into a secondary one ('disable').

This method has been introduced in version 0.27.3.

(2) Signature: void **enable_parameter** (string parameter_name, bool enable)

Description: Enables or disables a parameter.

Some parameters are 'secondary' parameters which are extracted but not handled in device compare and are not shown in the netlist browser. For example, the 'W' parameter of the resistor is such a secondary parameter. This method allows turning a parameter in a primary one ('enable') or into a secondary one ('disable').

This version accepts a parameter name.

This method has been introduced in version 0.27.3.

equal_parameters

Signature: EqualDeviceParameters ptr **equal_parameters**

Description: Gets the device parameter comparer for netlist verification or nil if no comparer is registered.

See equal_parameters= for the setter.

This method has been moved from 'GenericDeviceClass' to 'DeviceClass' in version 0.27.3.

Python specific notes:
The object exposes a readable attribute 'equal_parameters'. This is the getter.

equal_parameters=

Signature: void **equal_parameters=** (EqualDeviceParameters ptr comparer)

Description: Specifies a device parameter comparer for netlist verification.

By default, all devices are compared with all parameters. If you want to select only certain parameters for comparison or use a fuzzy compare criterion, use an EqualDeviceParameters object and assign it to the device class of one netlist. You can also chain multiple EqualDeviceParameters objects with the '+' operator for specifying multiple parameters in the equality check.

You can assign nil for the parameter comparer to remove it.

	In special cases, you can even implement a custom compare scheme by deriving your own comparer from the GenericDeviceParameterCompare class. This method has been moved from 'GenericDeviceClass' to 'DeviceClass' in version 0.27.3. **Python specific notes:** The object exposes a writable attribute 'equal_parameters'. This is the setter.
equivalent_terminal_id	**Signature**: void **equivalent_terminal_id** (unsigned long original_id, unsigned long equivalent_id) **Description**: Specifies a terminal to be equivalent to another. Use this method to specify two terminals to be exchangeable. For example to make S and D of a MOS transistor equivalent, call this method with S and D terminal IDs. In netlist matching, S will be translated to D and thus made equivalent to D. Note that terminal equivalence is not effective if the device class operates in strict mode (see DeviceClass#strict=). This method has been moved from 'GenericDeviceClass' to 'DeviceClass' in version 0.27.3.
has_parameter?	**Signature**: *[const]* bool **has_parameter?** (string name) **Description**: Returns true, if the device class has a parameter with the given name.
has_terminal?	**Signature**: *[const]* bool **has_terminal?** (string name) **Description**: Returns true, if the device class has a terminal with the given name.
id	**Signature**: *[const]* unsigned long **id** **Description**: Gets the unique ID of the device class The ID is a unique integer that identifies the device class. Use the ID to check for object identity - i.e. to determine whether two devices share the same device class.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
name	**Signature**: *[const]* string **name** **Description**: Gets the name of the device class. **Python specific notes:** The object exposes a readable attribute 'name'. This is the getter.
name=	**Signature**: void **name=** (string name) **Description**: Sets the name of the device class. **Python specific notes:** The object exposes a writable attribute 'name'. This is the setter.
netlist	**Signature**: Netlist ptr **netlist** **Description**: Gets the netlist the device class lives in.

new

Signature: *[static]* new DeviceClass ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

parameter_definition

(1) Signature: *[const]* const DeviceParameterDefinition ptr **parameter_definition** (unsigned long parameter_id)

Description: Gets the parameter definition object for a given ID.

Parameter definition IDs are used in some places to reference a specific parameter of a device. This method obtains the corresponding definition object.

(2) Signature: *[const]* const DeviceParameterDefinition ptr **parameter_definition** (string parameter_name)

Description: Gets the parameter definition object for a given ID.

Parameter definition IDs are used in some places to reference a specific parameter of a device. This method obtains the corresponding definition object. This version accepts a parameter name.

This method has been introduced in version 0.27.3.

parameter_definitions

Signature: *[const]* DeviceParameterDefinition[] **parameter_definitions**

Description: Gets the list of parameter definitions of the device.

See the DeviceParameterDefinition class description for details.

parameter_id

Signature: *[const]* unsigned long **parameter_id** (string name)

Description: Returns the parameter ID of the parameter with the given name.

An exception is thrown if there is no parameter with the given name. Use has_parameter to check whether the name is a valid parameter name.

strict=

Signature: void **strict=** (bool s)

Description: Sets a value indicating whether this class performs strict terminal mapping

Classes with this flag set never allow terminal swapping, even if the device symmetry supports that. If two classes are involved in a netlist compare, terminal swapping will be disabled if one of the classes is in strict mode.

By default, device classes are not strict and terminal swapping is allowed as far as the device symmetry supports that.

Python specific notes:
The object exposes a writable attribute 'strict'. This is the setter.

strict?

Signature: *[const]* bool **strict?**

Description: Gets a value indicating whether this class performs strict terminal mapping

See strict= for details about this attribute.

Python specific notes:
The object exposes a readable attribute 'strict'. This is the getter.

supports_parallel_combination=

Signature: void **supports_parallel_combination=** (bool f)

Description: Specifies whether the device supports parallel device combination.

Parallel device combination means that all terminals of two combination candidates are connected to the same nets. If the device does not support this combination mode, this predicate can be set

to false. This will make the device extractor skip the combination test in parallel mode and improve performance somewhat.

This method has been moved from 'GenericDeviceClass' to 'DeviceClass' in version 0.27.3.

Python specific notes:
The object exposes a writable attribute 'supports_parallel_combination'. This is the setter.

supports_serial_combination

Signature: void **supports_serial_combination=** (bool f)

Description: Specifies whether the device supports serial device combination.

Serial device combination means that the devices are connected by internal nodes. If the device does not support this combination mode, this predicate can be set to false. This will make the device extractor skip the combination test in serial mode and improve performance somewhat.

This method has been moved from 'GenericDeviceClass' to 'DeviceClass' in version 0.27.3.

Python specific notes:
The object exposes a writable attribute 'supports_serial_combination'. This is the setter.

terminal_definition

Signature: *[const]* const DeviceTerminalDefinition ptr **terminal_definition** (unsigned long terminal_id)

Description: Gets the terminal definition object for a given ID.

Terminal definition IDs are used in some places to reference a specific terminal of a device. This method obtains the corresponding definition object.

terminal_definitions

Signature: *[const]* DeviceTerminalDefinition[] **terminal_definitions**

Description: Gets the list of terminal definitions of the device.

See the DeviceTerminalDefinition class description for details.

terminal_id

Signature: *[const]* unsigned long **terminal_id** (string name)

Description: Returns the terminal ID of the terminal with the given name.

An exception is thrown if there is no terminal with the given name. Use has_terminal to check whether the name is a valid terminal name.

4.133. API reference - Class Circuit

Notation used in Ruby API documentation

Module: db

Description: Circuits are the basic building blocks of the netlist

Class hierarchy: Circuit » NetlistObject

A circuit has pins by which it can connect to the outside. Pins are created using create_pin and are represented by the Pin class.

Furthermore, a circuit manages the components of the netlist. Components are devices (class Device) and subcircuits (class SubCircuit). Devices are basic devices such as resistors or transistors. Subcircuits are other circuits to which nets from this circuit connect. Devices are created using the create_device method. Subcircuits are created using the create_subcircuit method.

Devices are connected through 'terminals', subcircuits are connected through their pins. Terminals and pins are described by integer ID's in the context of most methods.

Finally, the circuit consists of the nets. Nets connect terminals of devices and pins of subcircuits or the circuit itself. Nets are created using create_net and are represented by objects of the Net class. See there for more about nets.

The Circuit object is only valid if the netlist object is alive. Circuits must be added to a netlist using Netlist#add to become part of the netlist.

The Circuit class has been introduced in version 0.26.

Public methods

	void	**_assign**	(const Circuit other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new Circuit ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**blank**		Blanks out the circuit
[const]	DPolygon	**boundary**		Gets the boundary of the circuit
	void	**boundary=**	(const DPolygon boundary)	Sets the boundary of the circuit
[const]	unsigned int	**cell_index**		Gets the cell index of the circuit
	void	**cell_index=**	(unsigned int cell_index)	Sets the cell index
	void	**clear**		Clears the circuit

	void	**combine_devices**		Combines devices where possible
	void	**connect_pin**	(unsigned long pin_id, Net ptr net)	Connects the given pin with the given net.
	void	**connect_pin**	(const Pin ptr pin, Net ptr net)	Connects the given pin with the given net.
	Device ptr	**create_device**	(DeviceClass ptr device_class, string name =)	Creates a new bound Device object inside the circuit
	Net ptr	**create_net**	(string name =)	Creates a new Net object inside the circuit
	Pin ptr	**create_pin**	(string name)	Creates a new Pin object inside the circuit
	SubCircuit ptr	**create_subcircuit**	(Circuit ptr circuit, string name =)	Creates a new bound SubCircuit object inside the circuit
	Device ptr	**device_by_id**	(unsigned long id)	Gets the device object for a given ID.
[const]	const Device ptr	**device_by_id**	(unsigned long id)	Gets the device object for a given ID (const version).
	Device ptr	**device_by_name**	(string name)	Gets the device object for a given name.
[const]	const Device ptr	**device_by_name**	(string name)	Gets the device object for a given name (const version).
	void	**disconnect_pin**	(unsigned long pin_id)	Disconnects the given pin from any net.
	void	**disconnect_pin**	(const Pin ptr pin)	Disconnects the given pin from any net.
[const]	bool	**dont_purge**		Gets a value indicating whether the circuit can be purged on Netlist#purge.
	void	**dont_purge=**	(bool f)	Sets a value indicating whether the circuit can be purged on Netlist#purge.
[iter]	Circuit	**each_child**		Iterates over the child circuits of this circuit
[const,iter]	Circuit	**each_child**		Iterates over the child circuits of this circuit (const version)
[iter]	Device	**each_device**		Iterates over the devices of the circuit
[const,iter]	Device	**each_device**		Iterates over the devices of the circuit (const version)
[iter]	Net	**each_net**		Iterates over the nets of the circuit
[const,iter]	Net	**each_net**		Iterates over the nets of the circuit (const version)
[iter]	Circuit	**each_parent**		Iterates over the parent circuits of this circuit

[const,iter]	Circuit	**each_parent**		Iterates over the parent circuits of this circuit (const version)
[iter]	Pin	**each_pin**		Iterates over the pins of the circuit
[const,iter]	Pin	**each_pin**		Iterates over the pins of the circuit (const version)
[iter]	SubCircuit	**each_ref**		Iterates over the subcircuit objects referencing this circuit
[const,iter]	SubCircuit	**each_ref**		Iterates over the subcircuit objects referencing this circuit (const version)
[iter]	SubCircuit	**each_subcircuit**		Iterates over the subcircuits of the circuit
[const,iter]	SubCircuit	**each_subcircuit**		Iterates over the subcircuits of the circuit (const version)
	void	**flatten_subcircuit**	(SubCircuit ptr subcircuit)	Flattens a subcircuit
[const]	bool	**has_refs?**		Returns a value indicating whether the circuit has references
	void	**join_nets**	(Net ptr net, Net ptr with)	Joins (connects) two nets into one
[const]	string	**name**		Gets the name of the circuit
	void	**name=**	(string name)	Sets the name of the circuit
	Net ptr	**net_by_cluster_id**	(unsigned long cluster_id)	Gets the net object corresponding to a specific cluster ID
	Net ptr	**net_by_name**	(string name)	Gets the net object for a given name.
[const]	const Net ptr	**net_by_name**	(string name)	Gets the net object for a given name (const version).
	Net ptr	**net_for_pin**	(unsigned long pin_id)	Gets the net object attached to a specific pin.
[const]	const Net ptr	**net_for_pin**	(unsigned long pin_id)	Gets the net object attached to a specific pin (const version).
	Net ptr	**net_for_pin**	(const Pin ptr pin)	Gets the net object attached to a specific pin.
[const]	const Net ptr	**net_for_pin**	(const Pin ptr pin)	Gets the net object attached to a specific pin (const version).
	Netlist ptr	**netlist**		Gets the netlist object the circuit lives in
[const]	const Netlist ptr	**netlist**		Gets the netlist object the circuit lives in (const version)
	Net ptr[]	**nets_by_name**	(string name_pattern)	Gets the net objects for a given name filter.

[const]	const Net ptr[]	**nets_by_name**	(string name_pattern)	Gets the net objects for a given name filter (const version).
	Pin ptr	**pin_by_id**	(unsigned long id)	Gets the Pin object corresponding to a specific ID
[const]	const Pin ptr	**pin_by_id**	(unsigned long id)	Gets the Pin object corresponding to a specific ID (const version)
	Pin ptr	**pin_by_name**	(string name)	Gets the Pin object corresponding to a specific name
[const]	const Pin ptr	**pin_by_name**	(string name)	Gets the Pin object corresponding to a specific name (const version)
[const]	unsigned long	**pin_count**		Gets the number of pins in the circuit
	void	**purge_nets**		Purges floating nets.
	void	**purge_nets_keep_pins**		Purges floating nets but keep pins.
	void	**remove_device**	(Device ptr device)	Removes the given device from the circuit
	void	**remove_net**	(Net ptr net)	Removes the given net from the circuit
	void	**remove_pin**	(unsigned long id)	Removes the pin with the given ID from the circuit
	void	**remove_subcircuit**	(SubCircuit ptr subcircuit)	Removes the given subcircuit from the circuit
	void	**rename_pin**	(unsigned long id, string new_name)	Renames the pin with the given ID to 'new_name'
	SubCircuit ptr	**subcircuit_by_id**	(unsigned long id)	Gets the subcircuit object for a given ID.
[const]	const SubCircuit ptr	**subcircuit_by_id**	(unsigned long id)	Gets the subcircuit object for a given ID (const version).
	SubCircuit ptr	**subcircuit_by_name**	(string name)	Gets the subcircuit object for a given name.
[const]	const SubCircuit ptr	**subcircuit_by_name**	(string name)	Gets the subcircuit object for a given name (const version).

Detailed description

_assign

Signature: void **_assign** (const Circuit other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new Circuit ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

blank

Signature: void **blank**

Description: Blanks out the circuit

This method will remove all the innards of the circuit and just leave the pins. The pins won't be connected to inside nets anymore, but the circuit can still be called by subcircuit references. This method will eventually create a 'circuit abstract' (or black box). It will set the dont_purge flag to mark this circuit as 'intentionally empty'.

boundary

Signature: *[const]* DPolygon **boundary**

Description: Gets the boundary of the circuit

Python specific notes:

The object exposes a readable attribute 'boundary'. This is the getter.

boundary=

Signature: void **boundary=** (const DPolygon boundary)

Description: Sets the boundary of the circuit

Python specific notes:
The object exposes a writable attribute 'boundary'. This is the setter.

cell_index

Signature: *[const]* unsigned int **cell_index**

Description: Gets the cell index of the circuit

See cell_index= for details.

Python specific notes:
The object exposes a readable attribute 'cell_index'. This is the getter.

cell_index=

Signature: void **cell_index=** (unsigned int cell_index)

Description: Sets the cell index

The cell index relates a circuit with a cell from a layout. It's intended to hold a cell index number if the netlist was extracted from a layout.

Python specific notes:
The object exposes a writable attribute 'cell_index'. This is the setter.

clear

Signature: void **clear**

Description: Clears the circuit

This method removes all objects and clears the other attributes.

combine_devices

Signature: void **combine_devices**

Description: Combines devices where possible

This method will combine devices that can be combined according to their device classes 'combine_devices' method. For example, serial or parallel resistors can be combined into a single resistor.

connect_pin

(1) Signature: void **connect_pin** (unsigned long pin_id, Net ptr net)

Description: Connects the given pin with the given net.

The net must be one inside the circuit. Any previous connected is resolved before this connection is made. A pin can only be connected to one net at a time.

(2) Signature: void **connect_pin** (const Pin ptr pin, Net ptr net)

Description: Connects the given pin with the given net.

The net and the pin must be objects from inside the circuit. Any previous connected is resolved before this connection is made. A pin can only be connected to one net at a time.

create_device

Signature: Device ptr **create_device** (DeviceClass ptr device_class, string name =)

Description: Creates a new bound Device object inside the circuit

This object describes a device of the circuit. The device is already attached to the device class. The name is optional and is used to identify the device in a netlist file.

For more details see the Device class.

create_net

Signature: Net ptr **create_net** (string name =)

Description: Creates a new Net object inside the circuit

This object will describe a net of the circuit. The nets are basically connections between the different components of the circuit (subcircuits, devices and pins).

A net needs to be filled with references to connect to specific objects. See the Net class for more details.

create_pin

Signature: Pin ptr **create_pin** (string name)

Description: Creates a new Pin object inside the circuit

This object will describe a pin of the circuit. A circuit connects to the outside through such a pin. The pin is added after all existing pins. For more details see the Pin class.

Starting with version 0.26.8, this method returns a reference to a Pin object rather than a copy.

create_subcircuit

Signature: SubCircuit ptr **create_subcircuit** (Circuit ptr circuit, string name =)

Description: Creates a new bound SubCircuit object inside the circuit

This object describes an instance of another circuit inside the circuit. The subcircuit is already attached to the other circuit. The name is optional and is used to identify the subcircuit in a netlist file.

For more details see the SubCircuit class.

device_by_id

(1) Signature: Device ptr **device_by_id** (unsigned long id)

Description: Gets the device object for a given ID.

If the ID is not a valid device ID, nil is returned.

(2) Signature: *[const]* const Device ptr **device_by_id** (unsigned long id)

Description: Gets the device object for a given ID (const version).

If the ID is not a valid device ID, nil is returned.

This constness variant has been introduced in version 0.26.8

device_by_name

(1) Signature: Device ptr **device_by_name** (string name)

Description: Gets the device object for a given name.

If the ID is not a valid device name, nil is returned.

(2) Signature: *[const]* const Device ptr **device_by_name** (string name)

Description: Gets the device object for a given name (const version).

If the ID is not a valid device name, nil is returned.

This constness variant has been introduced in version 0.26.8

disconnect_pin

(1) Signature: void **disconnect_pin** (unsigned long pin_id)

Description: Disconnects the given pin from any net.

(2) Signature: void **disconnect_pin** (const Pin ptr pin)

Description: Disconnects the given pin from any net.

dont_purge

Signature: *[const]* bool **dont_purge**

Description: Gets a value indicating whether the circuit can be purged on Netlist#purge.

Python specific notes:
The object exposes a readable attribute 'dont_purge'. This is the getter.

dont_purge=

Signature: void **dont_purge=** (bool f)

Description: Sets a value indicating whether the circuit can be purged on Netlist#purge.

If this attribute is set to true, Netlist#purge will never delete this circuit. This flag therefore marks this circuit as 'precious'.

Python specific notes:
The object exposes a writable attribute 'dont_purge'. This is the setter.

each_child

(1) Signature: *[iter]* Circuit **each_child**

Description: Iterates over the child circuits of this circuit

Child circuits are the ones that are referenced from this circuit via subcircuits.

(2) Signature: *[const,iter]* Circuit **each_child**

Description: Iterates over the child circuits of this circuit (const version)

Child circuits are the ones that are referenced from this circuit via subcircuits.

This constness variant has been introduced in version 0.26.8

each_device

(1) Signature: *[iter]* Device **each_device**

Description: Iterates over the devices of the circuit

(2) Signature: *[const,iter]* Device **each_device**

Description: Iterates over the devices of the circuit (const version)

This constness variant has been introduced in version 0.26.8

each_net

(1) Signature: *[iter]* Net **each_net**

Description: Iterates over the nets of the circuit

(2) Signature: *[const,iter]* Net **each_net**

Description: Iterates over the nets of the circuit (const version)

This constness variant has been introduced in version 0.26.8

each_parent

(1) Signature: *[iter]* Circuit **each_parent**

Description: Iterates over the parent circuits of this circuit

Child circuits are the ones that are referencing this circuit via subcircuits.

(2) Signature: *[const,iter]* Circuit **each_parent**

Description: Iterates over the parent circuits of this circuit (const version)

Child circuits are the ones that are referencing this circuit via subcircuits.

This constness variant has been introduced in version 0.26.8

each_pin

(1) Signature: *[iter]* Pin **each_pin**
Description: Iterates over the pins of the circuit

(2) Signature: *[const,iter]* Pin **each_pin**
Description: Iterates over the pins of the circuit (const version)

This constness variant has been introduced in version 0.26.8

each_ref

(1) Signature: *[iter]* SubCircuit **each_ref**
Description: Iterates over the subcircuit objects referencing this circuit

(2) Signature: *[const,iter]* SubCircuit **each_ref**
Description: Iterates over the subcircuit objects referencing this circuit (const version)

This constness variant has been introduced in version 0.26.8

each_subcircuit

(1) Signature: *[iter]* SubCircuit **each_subcircuit**
Description: Iterates over the subcircuits of the circuit

(2) Signature: *[const,iter]* SubCircuit **each_subcircuit**
Description: Iterates over the subcircuits of the circuit (const version)

This constness variant has been introduced in version 0.26.8

flatten_subcircuit

Signature: void **flatten_subcircuit** (SubCircuit ptr subcircuit)
Description: Flattens a subcircuit

This method will substitute the given subcircuit by its contents. The subcircuit is removed after this.

has_refs?

Signature: *[const]* bool **has_refs?**
Description: Returns a value indicating whether the circuit has references

A circuit has references if there is at least one subcircuit referring to it.

join_nets

Signature: void **join_nets** (Net ptr net, Net ptr with)
Description: Joins (connects) two nets into one

This method will connect the 'with' net with 'net' and remove 'with'.

This method has been introduced in version 0.26.4. Starting with version 0.28.13, net names will be formed from both input names, combining them with as a comma-separated list.

name

Signature: *[const]* string **name**
Description: Gets the name of the circuit

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string name)
Description: Sets the name of the circuit

Python specific notes:

	The object exposes a writable attribute 'name'. This is the setter.
net_by_cluster_id	**Signature**: Net ptr **net_by_cluster_id** (unsigned long cluster_id) **Description**: Gets the net object corresponding to a specific cluster ID If the ID is not a valid pin cluster ID, nil is returned.
net_by_name	**(1) Signature**: Net ptr **net_by_name** (string name) **Description**: Gets the net object for a given name. If the ID is not a valid net name, nil is returned. **(2) Signature**: *[const]* const Net ptr **net_by_name** (string name) **Description**: Gets the net object for a given name (const version). If the ID is not a valid net name, nil is returned. This constness variant has been introduced in version 0.26.8
net_for_pin	**(1) Signature**: Net ptr **net_for_pin** (unsigned long pin_id) **Description**: Gets the net object attached to a specific pin. This is the net object inside the circuit which attaches to the given outward-bound pin. This method returns nil if the pin is not connected or the pin ID is invalid. **(2) Signature**: *[const]* const Net ptr **net_for_pin** (unsigned long pin_id) **Description**: Gets the net object attached to a specific pin (const version). This is the net object inside the circuit which attaches to the given outward-bound pin. This method returns nil if the pin is not connected or the pin ID is invalid. This constness variant has been introduced in version 0.26.8 **(3) Signature**: Net ptr **net_for_pin** (const Pin ptr pin) **Description**: Gets the net object attached to a specific pin. This is the net object inside the circuit which attaches to the given outward-bound pin. This method returns nil if the pin is not connected or the pin object is nil. **(4) Signature**: *[const]* const Net ptr **net_for_pin** (const Pin ptr pin) **Description**: Gets the net object attached to a specific pin (const version). This is the net object inside the circuit which attaches to the given outward-bound pin. This method returns nil if the pin is not connected or the pin object is nil. This constness variant has been introduced in version 0.26.8
netlist	**(1) Signature**: Netlist ptr **netlist** **Description**: Gets the netlist object the circuit lives in **(2) Signature**: *[const]* const Netlist ptr **netlist** **Description**: Gets the netlist object the circuit lives in (const version) This constness variant has been introduced in version 0.26.8

nets_by_name

(1) Signature: Net ptr[] **nets_by_name** (string name_pattern)

Description: Gets the net objects for a given name filter.

The name filter is a glob pattern. This method will return all Net objects matching the glob pattern.

This method has been introduced in version 0.27.3.

(2) Signature: *[const]* const Net ptr[] **nets_by_name** (string name_pattern)

Description: Gets the net objects for a given name filter (const version).

The name filter is a glob pattern. This method will return all Net objects matching the glob pattern.

This constness variant has been introduced in version 0.27.3

pin_by_id

(1) Signature: Pin ptr **pin_by_id** (unsigned long id)

Description: Gets the Pin object corresponding to a specific ID

If the ID is not a valid pin ID, nil is returned.

(2) Signature: *[const]* const Pin ptr **pin_by_id** (unsigned long id)

Description: Gets the Pin object corresponding to a specific ID (const version)

If the ID is not a valid pin ID, nil is returned.

This constness variant has been introduced in version 0.26.8

pin_by_name

(1) Signature: Pin ptr **pin_by_name** (string name)

Description: Gets the Pin object corresponding to a specific name

If the ID is not a valid pin name, nil is returned.

(2) Signature: *[const]* const Pin ptr **pin_by_name** (string name)

Description: Gets the Pin object corresponding to a specific name (const version)

If the ID is not a valid pin name, nil is returned.

This constness variant has been introduced in version 0.26.8

pin_count

Signature: *[const]* unsigned long **pin_count**

Description: Gets the number of pins in the circuit

purge_nets

Signature: void **purge_nets**

Description: Purges floating nets.

Floating nets are nets with no device or subcircuit attached to. Such floating nets are removed in this step. If these nets are connected outward to a circuit pin, this circuit pin is also removed.

purge_nets_keep_pins

Signature: void **purge_nets_keep_pins**

Description: Purges floating nets but keep pins.

This method will remove floating nets like purge_nets, but if these nets are attached to a pin, the pin will be left disconnected from any net.

This method has been introduced in version 0.26.2.

remove_device	**Signature**: void **remove_device** (Device ptr device) **Description**: Removes the given device from the circuit
remove_net	**Signature**: void **remove_net** (Net ptr net) **Description**: Removes the given net from the circuit
remove_pin	**Signature**: void **remove_pin** (unsigned long id) **Description**: Removes the pin with the given ID from the circuit This method has been introduced in version 0.26.2.
remove_subcircuit	**Signature**: void **remove_subcircuit** (SubCircuit ptr subcircuit) **Description**: Removes the given subcircuit from the circuit
rename_pin	**Signature**: void **rename_pin** (unsigned long id, string new_name) **Description**: Renames the pin with the given ID to 'new_name' This method has been introduced in version 0.26.8.
subcircuit_by_id	**(1) Signature**: SubCircuit ptr **subcircuit_by_id** (unsigned long id) **Description**: Gets the subcircuit object for a given ID. If the ID is not a valid subcircuit ID, nil is returned. **(2) Signature**: *[const]* const SubCircuit ptr **subcircuit_by_id** (unsigned long id) **Description**: Gets the subcircuit object for a given ID (const version). If the ID is not a valid subcircuit ID, nil is returned. This constness variant has been introduced in version 0.26.8
subcircuit_by_name	**(1) Signature**: SubCircuit ptr **subcircuit_by_name** (string name) **Description**: Gets the subcircuit object for a given name. If the ID is not a valid subcircuit name, nil is returned. **(2) Signature**: *[const]* const SubCircuit ptr **subcircuit_by_name** (string name) **Description**: Gets the subcircuit object for a given name (const version). If the ID is not a valid subcircuit name, nil is returned. This constness variant has been introduced in version 0.26.8

4.134. API reference - Class Netlist

Notation used in Ruby API documentation

Module: db

Description: The netlist top-level class

A netlist is a hierarchical structure of circuits. At least one circuit is the top-level circuit, other circuits may be referenced as subcircuits. Circuits are created with create_circuit and are represented by objects of the Circuit class.

Beside circuits, the netlist manages device classes. Device classes describe specific types of devices. Device classes are represented by objects of the DeviceClass class and are created using create_device_class.

The netlist class has been introduced with version 0.26.

Public constructors

new Netlist ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add**	(Circuit ptr circuit)	Adds the circuit to the netlist
	void	**add**	(DeviceClass ptr device_class)	Adds the device class to the netlist
	void	**assign**	(const Netlist other)	Assigns another object to self
	void	**blank_circuit**	(string pattern)	Blanks circuits matching a certain pattern
	void	**case_sensitive=**	(bool cs)	Sets a value indicating whether the netlist names are case sensitive
	Circuit ptr	**circuit_by_cell_index**	(unsigned int cell_index)	Gets the circuit object for a given cell index.
[const]	const Circuit ptr	**circuit_by_cell_index**	(unsigned int cell_index)	Gets the circuit object for a given cell index (const version).
	Circuit ptr	**circuit_by_name**	(string name)	Gets the circuit object for a given name.

[const]	const Circuit ptr	**circuit_by_name**	(string name)	Gets the circuit object for a given name (const version).
	Circuit ptr[]	**circuits_by_name**	(string name_pattern)	Gets the circuit objects for a given name filter.
[const]	const Circuit ptr[]	**circuits_by_name**	(string name_pattern)	Gets the circuit objects for a given name filter (const version).
	void	**combine_devices**		Combines devices where possible
	DeviceClass ptr	**device_class_by_name**	(string name)	Gets the device class for a given name.
[const]	const DeviceClass ptr	**device_class_by_name**	(string name)	Gets the device class for a given name (const version).
[const]	new Netlist ptr	**dup**		Creates a copy of self
[iter]	Circuit	**each_circuit**		Iterates over the circuits of the netlist
[const,iter]	Circuit	**each_circuit**		Iterates over the circuits of the netlist (const version)
[iter]	Circuit	**each_circuit_bottom_up**		Iterates over the circuits bottom-up
[const,iter]	Circuit	**each_circuit_bottom_up**		Iterates over the circuits bottom-up (const version)
[iter]	Circuit	**each_circuit_top_down**		Iterates over the circuits top-down
[const,iter]	Circuit	**each_circuit_top_down**		Iterates over the circuits top-down (const version)
[iter]	DeviceClass	**each_device_class**		Iterates over the device classes of the netlist
[const,iter]	DeviceClass	**each_device_class**		Iterates over the device classes of the netlist (const version)
	void	**flatten**		Flattens all circuits of the netlist
	void	**flatten_circuit**	(Circuit ptr circuit)	Flattens a subcircuit
	void	**flatten_circuit**	(string pattern)	Flattens circuits matching a certain pattern
	void	**flatten_circuits**	(Circuit ptr[] circuits)	Flattens all given circuits of the netlist
	void	**from_s**	(string str)	Reads the netlist from a string representation.
[const]	bool	**is_case_sensitive?**		Returns a value indicating whether the netlist names are case sensitive
	void	**make_top_level_pins**		Creates pins for top-level circuits.
	Net ptr[]	**nets_by_name**	(string name_pattern)	Gets the net objects for a given name filter.

[const]	const Net ptr[]	**nets_by_name**	(string name_pattern)	Gets the net objects for a given name filter (const version).
	void	**purge**		Purge unused nets, circuits and subcircuits.
	void	**purge_circuit**	(Circuit ptr circuit)	Removes the given circuit object and all child circuits which are not used otherwise from the netlist
	void	**purge_nets**		Purges floating nets.
	void	**read**	(string file, NetlistReader ptr reader)	Writes the netlist to the given file using the given reader object to parse the file
	void	**remove**	(Circuit ptr circuit)	Removes the given circuit object from the netlist
	void	**remove**	(DeviceClass ptr device_class)	Removes the given device class object from the netlist
	void	**simplify**		Convenience method that combines the simplification.
[const]	string	**to_s**		Converts the netlist to a string representation.
[const]	unsigned long	**top_circuit_count**		Gets the number of top circuits.
[const]	void	**write**	(string file, NetlistWriter ptr writer, string description =)	Writes the netlist to the given file using the given writer object to format the file

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add

(1) Signature: void **add** (Circuit ptr circuit)

Description: Adds the circuit to the netlist

This method will add the given circuit object to the netlist. After the circuit has been added, it will be owned by the netlist.

(2) Signature: void **add** (DeviceClass ptr device_class)

Description: Adds the device class to the netlist

This method will add the given device class object to the netlist. After the device class has been added, it will be owned by the netlist.

assign

Signature: void **assign** (const Netlist other)

Description: Assigns another object to self

blank_circuit

Signature: void **blank_circuit** (string pattern)

Description: Blanks circuits matching a certain pattern

This method will erase everything from inside the circuits matching the given pattern. It will only leave pins which are not connected to any net. Hence, this method forms 'abstract' or black-box circuits which can be instantiated through subcircuits like the former ones, but are empty shells. The name pattern is a glob expression. For example, 'blank_circuit("np*")' will blank out all circuits with names starting with 'np'.

For more details see Circuit#blank which is the corresponding method on the actual object.

case_sensitive=

Signature: void **case_sensitive=** (bool cs)

Description: Sets a value indicating whether the netlist names are case sensitive

This method has been added in version 0.27.3.

Python specific notes:
The object exposes a writable attribute 'case_sensitive'. This is the setter.

circuit_by_cell_index

(1) Signature: Circuit ptr **circuit_by_cell_index** (unsigned int cell_index)

Description: Gets the circuit object for a given cell index.

If the cell index is not valid or no circuit is registered with this index, nil is returned.

(2) Signature: *[const]* const Circuit ptr **circuit_by_cell_index** (unsigned int cell_index)

Description: Gets the circuit object for a given cell index (const version).

If the cell index is not valid or no circuit is registered with this index, nil is returned.

This constness variant has been introduced in version 0.26.8.

circuit_by_name

(1) Signature: Circuit ptr **circuit_by_name** (string name)

Description: Gets the circuit object for a given name.

If the name is not a valid circuit name, nil is returned.

(2) Signature: *[const]* const Circuit ptr **circuit_by_name** (string name)

Description: Gets the circuit object for a given name (const version).

If the name is not a valid circuit name, nil is returned.

This constness variant has been introduced in version 0.26.8.

circuits_by_name

(1) Signature: Circuit ptr[] **circuits_by_name** (string name_pattern)

Description: Gets the circuit objects for a given name filter.

The name filter is a glob pattern. This method will return all Circuit objects matching the glob pattern.

This method has been introduced in version 0.26.4.

(2) Signature: *[const]* const Circuit ptr[] **circuits_by_name** (string name_pattern)

Description: Gets the circuit objects for a given name filter (const version).

The name filter is a glob pattern. This method will return all Circuit objects matching the glob pattern.

This constness variant has been introduced in version 0.26.8.

combine_devices

Signature: void **combine_devices**

Description: Combines devices where possible

This method will combine devices that can be combined according to their device classes 'combine_devices' method. For example, serial or parallel resistors can be combined into a single resistor.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

device_class_by_name

(1) Signature: DeviceClass ptr **device_class_by_name** (string name)

Description: Gets the device class for a given name.

If the name is not a valid device class name, nil is returned.

(2) Signature: *[const]* const DeviceClass ptr **device_class_by_name** (string name)

Description: Gets the device class for a given name (const version).

If the name is not a valid device class name, nil is returned.

This constness variant has been introduced in version 0.26.8.

dup

Signature: *[const]* new Netlist ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_circuit

(1) Signature: *[iter]* Circuit **each_circuit**

Description: Iterates over the circuits of the netlist

(2) Signature: *[const,iter]* Circuit **each_circuit**

Description: Iterates over the circuits of the netlist (const version)

This constness variant has been introduced in version 0.26.8.

each_circuit_bottom_up

(1) Signature: *[iter]* Circuit **each_circuit_bottom_up**

Description: Iterates over the circuits bottom-up

Iterating bottom-up means the parent circuits come after the child circuits. This is the basically the reverse order as delivered by each_circuit_top_down.

(2) Signature: *[const,iter]* Circuit **each_circuit_bottom_up**

Description: Iterates over the circuits bottom-up (const version)

Iterating bottom-up means the parent circuits come after the child circuits. This is the basically the reverse order as delivered by each_circuit_top_down.

This constness variant has been introduced in version 0.26.8.

each_circuit_top_down

(1) Signature: *[iter]* Circuit **each_circuit_top_down**

Description: Iterates over the circuits top-down

Iterating top-down means the parent circuits come before the child circuits. The first top_circuit_count circuits are top circuits - i.e. those which are not referenced by other circuits.

(2) Signature: *[const,iter]* Circuit **each_circuit_top_down**

Description: Iterates over the circuits top-down (const version)

Iterating top-down means the parent circuits come before the child circuits. The first top_circuit_count circuits are top circuits - i.e. those which are not referenced by other circuits.

This constness variant has been introduced in version 0.26.8.

each_device_class

(1) Signature: *[iter]* DeviceClass **each_device_class**

Description: Iterates over the device classes of the netlist

(2) Signature: *[const,iter]* DeviceClass **each_device_class**

Description: Iterates over the device classes of the netlist (const version)

This constness variant has been introduced in version 0.26.8.

flatten

Signature: void **flatten**

Description: Flattens all circuits of the netlist

After calling this method, only the top circuits will remain.

flatten_circuit

(1) Signature: void **flatten_circuit** (Circuit ptr circuit)

Description: Flattens a subcircuit

This method will substitute all instances (subcircuits) of the given circuit by its contents. After this, the circuit is removed.

(2) Signature: void **flatten_circuit** (string pattern)

Description: Flattens circuits matching a certain pattern

This method will substitute all instances (subcircuits) of all circuits with names matching the given name pattern. The name pattern is a glob expression. For example, 'flatten_circuit("np*")' will flatten all circuits with names starting with 'np'.

flatten_circuits

Signature: void **flatten_circuits** (Circuit ptr[] circuits)

Description: Flattens all given circuits of the netlist

This method is equivalent to calling flatten_circuit for all given circuits, but more efficient.

This method has been introduced in version 0.26.1

from_s

Signature: void **from_s** (string str)

Description: Reads the netlist from a string representation.

This method is intended for test purposes mainly. It turns a string returned by to_s back into a netlist. Note that the device classes must be created before as they are not persisted inside the string.

is_case_sensitive?

Signature: *[const]* bool **is_case_sensitive?**

Description: Returns a value indicating whether the netlist names are case sensitive

This method has been added in version 0.27.3.

Python specific notes:
The object exposes a readable attribute 'case_sensitive'. This is the getter.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

make_top_level_pins

Signature: void **make_top_level_pins**

Description: Creates pins for top-level circuits.

This method will turn all named nets of top-level circuits (such that are not referenced by subcircuits) into pins. This method can be used before purge to avoid that purge will remove nets which are directly connecting to subcircuits.

nets_by_name

(1) Signature: Net ptr[] **nets_by_name** (string name_pattern)

Description: Gets the net objects for a given name filter.

The name filter is a glob pattern. This method will return all Net objects matching the glob pattern.

This method has been introduced in version 0.28.4.

(2) Signature: *[const]* const Net ptr[] **nets_by_name** (string name_pattern)

Description: Gets the net objects for a given name filter (const version).

The name filter is a glob pattern. This method will return all Net objects matching the glob pattern.

This constness variant has been introduced in version 0.28.4.

new

Signature: *[static]* new Netlist ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

purge

Signature: void **purge**

Description: Purge unused nets, circuits and subcircuits.

This method will purge all nets which return floating == true. Circuits which don't have any nets (or only floating ones) and removed. Their subcircuits are disconnected. This method respects the Circuit#dont_purge attribute and will never delete circuits with this flag set.

purge_circuit

Signature: void **purge_circuit** (Circuit ptr circuit)

Description: Removes the given circuit object and all child circuits which are not used otherwise from the netlist

After the circuit has been removed, the object becomes invalid and cannot be used further. A circuit with references (see has_refs?) should not be removed as the subcircuits calling it would afterwards point to nothing.

purge_nets

Signature: void **purge_nets**

Description: Purges floating nets.

Floating nets can be created as effect of reconnections of devices or pins. This method will eliminate all nets that make less than two connections.

read

Signature: void **read** (string file, NetlistReader ptr reader)

Description: Writes the netlist to the given file using the given reader object to parse the file

See NetlistSpiceReader for an example for a parser.

remove

(1) Signature: void **remove** (Circuit ptr circuit)

Description: Removes the given circuit object from the netlist

After the circuit has been removed, the object becomes invalid and cannot be used further. A circuit with references (see has_refs?) should not be removed as the subcircuits calling it would afterwards point to nothing.

(2) Signature: void **remove** (DeviceClass ptr device_class)

Description: Removes the given device class object from the netlist

After the object has been removed, it becomes invalid and cannot be used further. Use this method with care as it may corrupt the internal structure of the netlist. Only use this method when device refers to this device class.

simplify

Signature: void **simplify**

Description: Convenience method that combines the simplification.

This method is a convenience method that runs make_top_level_pins, purge, combine_devices and purge_nets.

to_s

Signature: *[const]* string **to_s**

Description: Converts the netlist to a string representation.

This method is intended for test purposes mainly.

Python specific notes:
This method is also available as 'str(object)'.

top_circuit_count

Signature: *[const]* unsigned long **top_circuit_count**

Description: Gets the number of top circuits.

Top circuits are those which are not referenced by other circuits via subcircuits. A well-formed netlist has a single top circuit.

write

Signature: *[const]* void **write** (string file, NetlistWriter ptr writer, string description =)

Description: Writes the netlist to the given file using the given writer object to format the file

See NetlistSpiceWriter for an example for a formatter. The description is an arbitrary text which will be put into the file somewhere at the beginning.

4.135. API reference - Class NetlistSpiceWriterDelegate

Notation used in Ruby API documentation

Module: db

Description: Provides a delegate for the SPICE writer for doing special formatting for devices

Supply a customized class to provide a specialized writing scheme for devices. You need a customized class if you want to implement special devices or you want to use subcircuits rather than the built-in devices.

See NetlistSpiceWriter for more details.

This class has been introduced in version 0.26.

Public constructors

new NetlistSpiceWriterDelegate ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetlistSpiceWriterD other)	Assigns another object to self
[const]	new NetlistSpiceWriterDelegate ptr	**dup**		Creates a copy of self
[const]	void	**emit_comment**	(string comment)	Writes the given comment into the file
[const]	void	**emit_line**	(string line)	Writes the given line into the file
[const]	string	**format_name**	(string name)	Formats the given name in a SPICE-compatible way
[const]	string	**net_to_string**	(const Net ptr net)	Gets the node ID for the given net
[virtual,const]	void	**write_device**	(Device device)	Inserts a text for the given device

[virtual,const]	void	**write_device_intro**	(DeviceClass device_class)	Inserts a text for the given device class
[virtual,const]	void	**write_header**		Writes the text at the beginning of the SPICE netlist

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const NetlistSpiceWriterDelegate other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetlistSpiceWriterDelegate ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

emit_comment

Signature: *[const]* void **emit_comment** (string comment)

Description: Writes the given comment into the file

emit_line

Signature: *[const]* void **emit_line** (string line)

Description: Writes the given line into the file

format_name

Signature: *[const]* string **format_name** (string name)

Description: Formats the given name in a SPICE-compatible way

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

net_to_string

Signature: *[const]* string **net_to_string** (const Net ptr net)

Description: Gets the node ID for the given net

The node ID is a numeric string instead of the full name of the net. Numeric IDs are used within SPICE netlist because they are usually shorter.

new

Signature: *[static]* new NetlistSpiceWriterDelegate ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

write_device

Signature: *[virtual,const]* void **write_device** (Device device)

Description: Inserts a text for the given device

Reimplement this method to write the given device in the desired way. The default implementation will utilize the device class information to write native SPICE elements for the devices.

write_device_intro

Signature: *[virtual,const]* void **write_device_intro** (DeviceClass device_class)

Description: Inserts a text for the given device class

Reimplement this method to insert your own text at the beginning of the file for the given device class

write_header

Signature: *[virtual,const]* void **write_header**

Description: Writes the text at the beginning of the SPICE netlist

Reimplement this method to insert your own text at the beginning of the file

4.136. API reference - Class NetlistWriter

Notation used in Ruby API documentation

Module: db

Description: Base class for netlist writers

This class is provided as a base class for netlist writers. It is not intended for reimplementation on script level, but used internally as an interface.

This class has been introduced in version 0.26.

Public constructors

new NetlistWriter ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new NetlistWriter ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

4.137. API reference - Class NetlistSpiceWriter

Notation used in Ruby API documentation

Module: db

Description: Implements a netlist writer for the SPICE format.

Class hierarchy: NetlistSpiceWriter » NetlistWriter

Provide a delegate for customizing the way devices are written.

Use the SPICE writer like this:

```
writer = RBA::NetlistSpiceWriter::new
netlist.write(path, writer)
```

You can give a custom description for the headline:

```
writer = RBA::NetlistSpiceWriter::new
netlist.write(path, writer, "A custom description")
```

To customize the output, you can use a device writer delegate. The delegate is an object of a class derived from NetlistSpiceWriterDelegate which reimplements several methods to customize the following parts:

- A global header (NetlistSpiceWriterDelegate#write_header): this method is called to print the part right after the headline
- A per-device class header (NetlistSpiceWriterDelegate#write_device_intro): this method is called for every device class and may print device-class specific headers (e.g. model definitions)
- Per-device output: this method (NetlistSpiceWriterDelegate#write_device): this method is called for every device and may print the device statement(s) in a specific way.

The delegate must use NetlistSpiceWriterDelegate#emit_line to print a line, NetlistSpiceWriterDelegate#emit_comment to print a comment etc. For more method see NetlistSpiceWriterDelegate.

A sample with a delegate is this:

```
class MyDelegate < RBA::NetlistSpiceWriterDelegate

  def write_header
    emit_line("*** My special header")
  end

  def write_device_intro(cls)
    emit_comment("My intro for class " + cls.name)
  end

  def write_device(dev)
    if dev.device_class.name != "MYDEVICE"
      emit_comment("Terminal #1: " + net_to_string(dev.net_for_terminal(0)))
      emit_comment("Terminal #2: " + net_to_string(dev.net_for_terminal(1)))
      super(dev)
      emit_comment("After device " + dev.expanded_name)
    else
      super(dev)
    end
  end

end
```

```
# write the netlist with delegate:
writer = RBA::NetlistSpiceWriter::new(MyDelegate::new)
netlist.write(path, writer)
```

This class has been introduced in version 0.26.

Public constructors

new NetlistSpiceWriter ptr	**new**		Creates a new writer without delegate.
new NetlistSpiceWriter ptr	**new**	(NetlistSpiceWriterDelegate ptr delegate)	Creates a new writer with a delegate.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetlistSpice other)	Assigns another object to self
[const]	new NetlistSpiceWriter ptr	**dup**		Creates a copy of self
	void	**use_net_names=**	(bool f)	Sets a value indicating whether to use net names (true) or net numbers (false).
[const]	bool	**use_net_names?**		Gets a value indicating whether to use net names (true) or net numbers (false).
	void	**with_comments=**	(bool f)	Sets a value indicating whether to embed comments for position etc. (true) or not (false).
[const]	bool	**with_comments?**		Gets a value indicating whether to embed comments for position etc. (true) or not (false).

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

	Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
assign	**Signature**: void **assign** (const NetlistSpiceWriter other) **Description**: Assigns another object to self
dup	**Signature**: *[const]* new NetlistSpiceWriter ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
new	**(1) Signature**: *[static]* new NetlistSpiceWriter ptr **new** **Description**: Creates a new writer without delegate.

	Python specific notes: This method is the default initializer of the object. **(2) Signature**: *[static]* new NetlistSpiceWriter ptr **new** (NetlistSpiceWriterDelegate ptr delegate) **Description**: Creates a new writer with a delegate. **Python specific notes:** This method is the default initializer of the object.
use_net_names=	**Signature**: void **use_net_names=** (bool f) **Description**: Sets a value indicating whether to use net names (true) or net numbers (false). The default is to use net numbers. **Python specific notes:** The object exposes a writable attribute 'use_net_names'. This is the setter.
use_net_names?	**Signature**: *[const]* bool **use_net_names?** **Description**: Gets a value indicating whether to use net names (true) or net numbers (false). **Python specific notes:** The object exposes a readable attribute 'use_net_names'. This is the getter.
with_comments=	**Signature**: void **with_comments=** (bool f) **Description**: Sets a value indicating whether to embed comments for position etc. (true) or not (false). The default is to embed comments. **Python specific notes:** The object exposes a writable attribute 'with_comments'. This is the setter.
with_comments?	**Signature**: *[const]* bool **with_comments?** **Description**: Gets a value indicating whether to embed comments for position etc. (true) or not (false). **Python specific notes:** The object exposes a readable attribute 'with_comments'. This is the getter.

4.138. API reference - Class NetlistReader

Notation used in Ruby API documentation

Module: db

Description: Base class for netlist readers

This class is provided as a base class for netlist readers. It is not intended for reimplementation on script level, but used internally as an interface.

This class has been introduced in version 0.26.

Public constructors

new NetlistReader ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

	Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**Signature**: *[static]* new NetlistReader ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.

4.139. API reference - Class ParseElementComponentsData

Notation used in Ruby API documentation

Module: db

Description: Supplies the return value for \NetlistSpiceReaderDelegate#parse_element_components.

This is a structure with two members: 'strings' for the string arguments and 'parameters' for the named arguments.

This helper class has been introduced in version 0.27.1. Starting with version 0.28.6, named parameters can be string types too.

Public constructors

new ParseElementComponentsData ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const ParseElementC other)	Assigns another object to self
[const]	new ParseElementComponentsData ptr	**dup**		Creates a copy of self
[const]	map<string,variant>	**parameters**		Gets the (named) parameters
	void	**parameters=**	(map<string,variant> dict)	Sets the (named) parameters
[const]	string[]	**strings**		Gets the (unnamed) string parameters
	void	**strings=**	(string[] list)	Sets the (unnamed) string parameters

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead

[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const ParseElementComponentsData other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new ParseElementComponentsData ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new ParseElementComponentsData ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

parameters

Signature: *[const]* map<string,variant> **parameters**

Description: Gets the (named) parameters

Named parameters are typically (but not neccessarily) numerical, like 'w=0.15u'.

Python specific notes:
The object exposes a readable attribute 'parameters'. This is the getter.

parameters=

Signature: void **parameters=** (map<string,variant> dict)

Description: Sets the (named) parameters

Python specific notes:
The object exposes a writable attribute 'parameters'. This is the setter.

strings

Signature: *[const]* string[] **strings**

Description: Gets the (unnamed) string parameters

These parameters are typically net names or model name.

Python specific notes:
The object exposes a readable attribute 'strings'. This is the getter.

strings=

Signature: void **strings=** (string[] list)

Description: Sets the (unnamed) string parameters

Python specific notes:
The object exposes a writable attribute 'strings'. This is the setter.

4.140. API reference - Class ParseElementData

Notation used in Ruby API documentation

Module: db

Description: Supplies the return value for \NetlistSpiceReaderDelegate#parse_element.

This is a structure with four members: 'model_name' for the model name, 'value' for the default numerical value, 'net_names' for the net names and 'parameters' for the named parameters.

This helper class has been introduced in version 0.27.1. Starting with version 0.28.6, named parameters can be string types too.

Public constructors

new ParseElementData ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const ParseElementD other)	Assigns another object to self
[const]	new ParseElementData ptr	**dup**		Creates a copy of self
[const]	string	**model_name**		Gets the model name
	void	**model_name=**	(string m)	Sets the model name
[const]	string[]	**net_names**		Gets the net names
	void	**net_names=**	(string[] list)	Sets the net names
[const]	map<string,variant>	**parameters**		Gets the (named) parameters
	void	**parameters=**	(map<string,variant> dict)	Sets the (named) parameters
[const]	double	**value**		Gets the value
	void	**value=**	(double v)	Sets the value

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const ParseElementData other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new ParseElementData ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

model_name

Signature: *[const]* string **model_name**

Description: Gets the model name

Python specific notes:
The object exposes a readable attribute 'model_name'. This is the getter.

model_name=

Signature: void **model_name=** (string m)

Description: Sets the model name

Python specific notes:

	The object exposes a writable attribute 'model_name'. This is the setter.
net_names	**Signature**: *[const]* string[] **net_names** **Description**: Gets the net names **Python specific notes:** The object exposes a readable attribute 'net_names'. This is the getter.
net_names=	**Signature**: void **net_names=** (string[] list) **Description**: Sets the net names **Python specific notes:** The object exposes a writable attribute 'net_names'. This is the setter.
new	**Signature**: *[static]* new ParseElementData ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
parameters	**Signature**: *[const]* map<string,variant> **parameters** **Description**: Gets the (named) parameters **Python specific notes:** The object exposes a readable attribute 'parameters'. This is the getter.
parameters=	**Signature**: void **parameters=** (map<string,variant> dict) **Description**: Sets the (named) parameters **Python specific notes:** The object exposes a writable attribute 'parameters'. This is the setter.
value	**Signature**: *[const]* double **value** **Description**: Gets the value **Python specific notes:** The object exposes a readable attribute 'value'. This is the getter.
value=	**Signature**: void **value=** (double v) **Description**: Sets the value **Python specific notes:** The object exposes a writable attribute 'value'. This is the setter.

4.141. API reference - Class NetlistSpiceReaderDelegate

Notation used in Ruby API documentation

Module: db

Description: Provides a delegate for the SPICE reader for translating device statements

Supply a customized class to provide a specialized reading scheme for devices. You need a customized class if you want to implement device reading from model subcircuits or to translate device parameters.

See NetlistSpiceReader for more details.

This class has been introduced in version 0.26.

Public constructors

new NetlistSpiceReaderDelegate ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	void	**apply_parameter_scal**	(Device ptr device)	Applies parameter scaling to the given device
	void	**assign**	(const NetlistSpiceReaderDelegate other)	Assigns another object to self
[virtual]	bool	**control_statement**	(string line)	Receives control statements not understood by the standard reader
[const]	new NetlistSpiceReaderDelegate ptr	**dup**		Creates a copy of self
[virtual]	bool	**element**	(Circuit ptr circuit, string element, string name, string model, double value, Net ptr[] nets, map<string,variant> parameters)	Makes a device from an element line

	void	**error**	(string msg)	Issues an error with the given message.
[virtual]	void	**finish**	(Netlist ptr netlist)	This method is called when the reader is done reading a netlist successfully
[const]	double	**get_scale**		Gets the scale factor set with '.options scale=...'
[virtual]	ParseElementData	**parse_element**	(string s, string element)	Parses an element card
	ParseElementComponentsData	**parse_element_components**	(string s, map<string,variant> variables = {})	Parses a string into string and parameter components.
[virtual]	void	**start**	(Netlist ptr netlist)	This method is called when the reader starts reading a netlist
[virtual]	string	**translate_net_name**	(string net_name)	Translates a net name from the raw net name to the true net name
	variant	**value_from_string**	(string s, map<string,variant> variables = {})	Translates a string into a value
[const]	map<string,variant>	**variables**		Gets the variables defined inside the SPICE file during execution of 'parse_element'
[virtual]	bool	**wants_subcircuit**	(string circuit_name)	Returns true, if the delegate wants subcircuit elements with this name

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

apply_parameter_scaling

Signature: *[const]* void **apply_parameter_scaling** (Device ptr device)

Description: Applies parameter scaling to the given device

Applies SI scaling (according to the parameter's si_scaling attribute) and geometry scaling (according to the parameter's geo_scale_exponent attribute) to the device parameters. Use this method of finish the device when you have created a custom device yourself.

The geometry scale is taken from the '.options scale=...' control statement.

This method has been introduced in version 0.28.6.

assign

Signature: void **assign** (const NetlistSpiceReaderDelegate other)

Description: Assigns another object to self

control_statement

Signature: *[virtual]* bool **control_statement** (string line)

Description: Receives control statements not understood by the standard reader

When the reader encounters a control statement not understood by the parser, it will pass the line to the delegate using this method. The delegate can decide if it wants to read this statement. It should return true in this case.

This method has been introduced in version 0.27.1

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetlistSpiceReaderDelegate ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

element

Signature: *[virtual]* bool **element** (Circuit ptr circuit, string element, string name, string model, double value, Net ptr[] nets, map<string,variant> parameters)

Description: Makes a device from an element line

circuit:	The circuit that is currently read.
element:	The upper-case element code ("M", "R", ...).
name:	The element's name.
model:	The upper-case model name (may be empty).
value:	The default value (e.g. resistance for resistors) and may be zero.
nets:	The nets given in the element line.
parameters:	The parameters of the element statement (parameter names are upper case).

The default implementation will create corresponding devices for some known elements using the Spice writer's parameter conventions.

The method must return true, if the element was was understood and false otherwise.

Starting with version 0.28.6, the parameter values can be strings too.

error

Signature: void **error** (string msg)

Description: Issues an error with the given message.

Use this method to generate an error.

finish

Signature: *[virtual]* void **finish** (Netlist ptr netlist)

Description: This method is called when the reader is done reading a netlist successfully

get_scale

Signature: *[const]* double **get_scale**

Description: Gets the scale factor set with '.options scale=...'

This method has been introduced in version 0.28.6.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new NetlistSpiceReaderDelegate ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

parse_element

Signature: *[virtual]* ParseElementData **parse_element** (string s, string element)

Description: Parses an element card

s:	The specification part of the element line (the part after element code and name).
element:	The upper-case element code ("M", "R", ...).
Returns:	A ParseElementData object with the parts of the element.

This method receives a string with the element specification and the element code. It is supposed to parse the element line and return a model name, a value, a list of net names and a parameter value dictionary.

'parse_element' is called on every element card. The results of this call go into the element method to actually create the device. This method can be reimplemented to support other flavors of SPICE.

This method has been introduced in version 0.27.1

parse_element_components

Signature: ParseElementComponentsData **parse_element_components** (string s, map<string,variant> variables = {})

Description: Parses a string into string and parameter components.

This method is provided to simplify the implementation of 'parse_element'. It takes a string and splits it into string arguments and parameter values. For example, 'a b c=6' renders two string arguments in 'nn' and one parameter ('C'->6.0). It returns data ParseElementComponentsData object with the strings and parameters. The parameter names are already translated to upper case.

The variables dictionary defines named variables with the given values.

This method has been introduced in version 0.27.1. The variables argument has been added in version 0.28.6.

start

Signature: *[virtual]* void **start** (Netlist ptr netlist)

Description: This method is called when the reader starts reading a netlist

translate_net_name

Signature: *[virtual]* string **translate_net_name** (string net_name)

Description: Translates a net name from the raw net name to the true net name

The default implementation will replace backslash sequences by the corresponding character. 'translate_net_name' is called before a net name is turned into a net object. The method can be reimplemented to supply a different translation scheme for net names. For example, to translate special characters.

This method has been introduced in version 0.27.1

value_from_string

Signature: variant **value_from_string** (string s, map<string,variant> variables = {})

Description: Translates a string into a value

This function simplifies the implementation of SPICE readers by providing a translation of a unit-annotated string into double values. For example, '1k' is translated to 1000.0. In addition, simple formula evaluation is supported, e.g '(1+3)*2' is translated into 8.0.

The variables dictionary defines named variables with the given values.

This method has been introduced in version 0.27.1. The variables argument has been added in version 0.28.6.

variables

Signature: *[const]* map<string,variant> **variables**

Description: Gets the variables defined inside the SPICE file during execution of 'parse_element'

In order to evaluate formulas, this method allows accessing the variables that are present during the execution of the SPICE reader.

This method has been introduced in version 0.28.6.

wants_subcircuit

Signature: *[virtual]* bool **wants_subcircuit** (string circuit_name)

Description: Returns true, if the delegate wants subcircuit elements with this name

The name is always upper case.

4.142. API reference - Class NetlistSpiceReader

Notation used in Ruby API documentation

Module: db

Description: Implements a netlist Reader for the SPICE format.

Class hierarchy: NetlistSpiceReader » NetlistReader

Use the SPICE reader like this:

```
reader = RBA::NetlistSpiceReader::new
netlist = RBA::Netlist::new
netlist.read(path, reader)
```

The translation of SPICE elements can be tailored by providing a NetlistSpiceReaderDelegate class. This allows translating of device parameters and mapping of some subcircuits to devices.

The following example is a delegate that turns subcircuits called HVNMOS and HVPMOS into MOS4 devices with the parameters scaled by 1.5:

```
class MyDelegate < RBA::NetlistSpiceReaderDelegate

  # says we want to catch these subcircuits as devices
  def wants_subcircuit(name)
    name == "HVNMOS" || name == "HVPMOS"
  end

  # translate the element
  def element(circuit, el, name, model, value, nets, params)

    if el != "X"
      # all other elements are left to the standard implementation
      return super
    end

    if nets.size != 4
      error("Subcircuit #{model} needs four nodes")
    end

    # provide a device class
    cls = circuit.netlist.device_class_by_name(model)
    if ! cls
      cls = RBA::DeviceClassMOS4Transistor::new
      cls.name = model
      circuit.netlist.add(cls)
    end

    # create a device
    device = circuit.create_device(cls, name)

    # and configure the device
    [ "S", "G", "D", "B" ].each_with_index do |t,index|
      device.connect_terminal(t, nets[index])
    end
    params.each do |p,value|
      device.set_parameter(p, value * 1.5)
    end

  end

end
```

```
# usage:

mydelegate = MyDelegate::new
reader = RBA::NetlistSpiceReader::new(mydelegate)

nl = RBA::Netlist::new
nl.read(input_file, reader)
```

A somewhat contrived example for using the delegate to translate net names is this:

```
class MyDelegate < RBA::NetlistSpiceReaderDelegate

  # translates 'VDD' to 'VXX' and leave all other net names as is:
  alias translate_net_name_org translate_net_name
  def translate_net_name(n)
    return n == "VDD" ? "VXX" : translate_net_name_org(n)}
  end

end
```

This class has been introduced in version 0.26. It has been extended in version 0.27.1.

Public constructors

new NetlistSpiceReader ptr	**new**		Creates a new reader.
new NetlistSpiceReader ptr	**new**	(NetlistSpiceReaderDelegate ptr delegate)	Creates a new reader with a delegate.

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

new

(1) Signature: *[static]* new NetlistSpiceReader ptr **new**

Description: Creates a new reader.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new NetlistSpiceReader ptr **new** (NetlistSpiceReaderDelegate ptr delegate)

Description: Creates a new reader with a delegate.

Python specific notes:
This method is the default initializer of the object.

4.143. API reference - Class DeviceClassResistor

Notation used in Ruby API documentation

Module: db

Description: A device class for a resistor.

Class hierarchy: DeviceClassResistor » DeviceClass

This class describes a resistor. Resistors are defined by their combination behavior and the basic parameter 'R' which is the resistance in Ohm.

A resistor has two terminals, A and B. The parameters of a resistor are R (the value in Ohms), L and W (length and width in micrometers) and A and P (area and perimeter in square micrometers and micrometers respectively).

This class has been introduced in version 0.26.

Public methods

	void	**_assign**	(const DeviceClas other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new DeviceClassResistor ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

Public static methods and constants

[static,const]	unsigned long	**PARAM_A**	A constant giving the parameter ID for parameter A
[static,const]	unsigned long	**PARAM_L**	A constant giving the parameter ID for parameter L
[static,const]	unsigned long	**PARAM_P**	A constant giving the parameter ID for parameter P
[static,const]	unsigned long	**PARAM_R**	A constant giving the parameter ID for parameter R
[static,const]	unsigned long	**PARAM_W**	A constant giving the parameter ID for parameter W

[static,const]	unsigned long	**TERMINAL_A**	A constant giving the terminal ID for terminal A
[static,const]	unsigned long	**TERMINAL_B**	A constant giving the terminal ID for terminal B

Detailed description

PARAM_A

Signature: *[static,const]* unsigned long **PARAM_A**

Description: A constant giving the parameter ID for parameter A

Python specific notes:
The object exposes a readable attribute 'PARAM_A'. This is the getter.

PARAM_L

Signature: *[static,const]* unsigned long **PARAM_L**

Description: A constant giving the parameter ID for parameter L

Python specific notes:
The object exposes a readable attribute 'PARAM_L'. This is the getter.

PARAM_P

Signature: *[static,const]* unsigned long **PARAM_P**

Description: A constant giving the parameter ID for parameter P

Python specific notes:
The object exposes a readable attribute 'PARAM_P'. This is the getter.

PARAM_R

Signature: *[static,const]* unsigned long **PARAM_R**

Description: A constant giving the parameter ID for parameter R

Python specific notes:
The object exposes a readable attribute 'PARAM_R'. This is the getter.

PARAM_W

Signature: *[static,const]* unsigned long **PARAM_W**

Description: A constant giving the parameter ID for parameter W

Python specific notes:
The object exposes a readable attribute 'PARAM_W'. This is the getter.

TERMINAL_A

Signature: *[static,const]* unsigned long **TERMINAL_A**

Description: A constant giving the terminal ID for terminal A

Python specific notes:
The object exposes a readable attribute 'TERMINAL_A'. This is the getter.

TERMINAL_B

Signature: *[static,const]* unsigned long **TERMINAL_B**

Description: A constant giving the terminal ID for terminal B

Python specific notes:
The object exposes a readable attribute 'TERMINAL_B'. This is the getter.

_assign

Signature: void **_assign** (const DeviceClassResistor other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

	Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_dup	**Signature**: *[const]* new DeviceClassResistor ptr **_dup** **Description**: Creates a copy of self
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.

4.144. API reference - Class DeviceClassResistorWithBulk

Notation used in Ruby API documentation

Module: db

Description: A device class for a resistor with a bulk terminal (substrate, well).

Class hierarchy: DeviceClassResistorWithBulk » DeviceClassResistor » DeviceClass

This class is similar to DeviceClassResistor, but provides an additional terminal (BULK) for the well or substrate the resistor is embedded in.

The additional terminal is 'W' for the well/substrate terminal.

This class has been introduced in version 0.26.

Public methods

	void	**_assign**	(const DeviceClass other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new DeviceClassResistorWith ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

Public static methods and constants

[static,const]	unsigned long	**TERMINAL_W**	A constant giving the terminal ID for terminal W (well, bulk)

Detailed description

TERMINAL_W

Signature: *[static,const]* unsigned long **TERMINAL_W**

Description: A constant giving the terminal ID for terminal W (well, bulk)

Python specific notes:
The object exposes a readable attribute 'TERMINAL_W'. This is the getter.

_assign

Signature: void **_assign** (const DeviceClassResistorWithBulk other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new DeviceClassResistorWithBulk ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

4.145. API reference - Class DeviceClassCapacitor

Notation used in Ruby API documentation

Module: db

Description: A device class for a capacitor.

Class hierarchy: DeviceClassCapacitor » DeviceClass

This describes a capacitor. Capacitors are defined by their combination behavior and the basic parameter 'C' which is the capacitance in Farad.

A capacitor has two terminals, A and B. The parameters of a capacitor are C (the value in Farad) and A and P (area and perimeter in square micrometers and micrometers respectively).

This class has been introduced in version 0.26.

Public methods

	void	**_assign**	(const DeviceClas other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new DeviceClassCapacitor ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

Public static methods and constants

[static,const]	unsigned long	**PARAM_A**	A constant giving the parameter ID for parameter A
[static,const]	unsigned long	**PARAM_C**	A constant giving the parameter ID for parameter C
[static,const]	unsigned long	**PARAM_P**	A constant giving the parameter ID for parameter P
[static,const]	unsigned long	**TERMINAL_A**	A constant giving the terminal ID for terminal A
[static,const]	unsigned long	**TERMINAL_B**	A constant giving the terminal ID for terminal B

Detailed description

PARAM_A	**Signature**: *[static,const]* unsigned long **PARAM_A** **Description**: A constant giving the parameter ID for parameter A **Python specific notes:** The object exposes a readable attribute 'PARAM_A'. This is the getter.
PARAM_C	**Signature**: *[static,const]* unsigned long **PARAM_C** **Description**: A constant giving the parameter ID for parameter C **Python specific notes:** The object exposes a readable attribute 'PARAM_C'. This is the getter.
PARAM_P	**Signature**: *[static,const]* unsigned long **PARAM_P** **Description**: A constant giving the parameter ID for parameter P **Python specific notes:** The object exposes a readable attribute 'PARAM_P'. This is the getter.
TERMINAL_A	**Signature**: *[static,const]* unsigned long **TERMINAL_A** **Description**: A constant giving the terminal ID for terminal A **Python specific notes:** The object exposes a readable attribute 'TERMINAL_A'. This is the getter.
TERMINAL_B	**Signature**: *[static,const]* unsigned long **TERMINAL_B** **Description**: A constant giving the terminal ID for terminal B **Python specific notes:** The object exposes a readable attribute 'TERMINAL_B'. This is the getter.
_assign	**Signature**: void **_assign** (const DeviceClassCapacitor other) **Description**: Assigns another object to self
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed

	This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_dup	**Signature**: *[const]* new DeviceClassCapacitor ptr **_dup** **Description**: Creates a copy of self
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.

4.146. API reference - Class DeviceClassCapacitorWithBulk

Notation used in Ruby API documentation

Module: db

Description: A device class for a capacitor with a bulk terminal (substrate, well).

Class hierarchy: DeviceClassCapacitorWithBulk » DeviceClassCapacitor » DeviceClass

This class is similar to DeviceClassCapacitor, but provides an additional terminal (BULK) for the well or substrate the capacitor is embedded in.

The additional terminal is 'W' for the well/substrate terminal.

This class has been introduced in version 0.26.

Public methods

	void	**_assign**	(const DeviceClass other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new DeviceClassCapacitorWit ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

Public static methods and constants

[static,const]	unsigned long	**TERMINAL_W**	A constant giving the terminal ID for terminal W (well, bulk)

Detailed description

TERMINAL_W

Signature: *[static,const]* unsigned long **TERMINAL_W**

Description: A constant giving the terminal ID for terminal W (well, bulk)

Python specific notes:
The object exposes a readable attribute 'TERMINAL_W'. This is the getter.

_assign

Signature: void **_assign** (const DeviceClassCapacitorWithBulk other)

Description: Assigns another object to self

_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_dup	**Signature**: *[const]* new DeviceClassCapacitorWithBulk ptr **_dup** **Description**: Creates a copy of self
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.

4.147. API reference - Class DeviceClassInductor

Notation used in Ruby API documentation

Module: db

Description: A device class for an inductor.

Class hierarchy: DeviceClassInductor » DeviceClass

This class describes an inductor. Inductors are defined by their combination behavior and the basic parameter 'L' which is the inductance in Henry.

An inductor has two terminals, A and B.

This class has been introduced in version 0.26.

Public methods

	void	**_assign**	(const DeviceClas other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new DeviceClassInductor ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

Public static methods and constants

[static,const]	unsigned long	**PARAM_L**	A constant giving the parameter ID for parameter L
[static,const]	unsigned long	**TERMINAL_A**	A constant giving the terminal ID for terminal A
[static,const]	unsigned long	**TERMINAL_B**	A constant giving the terminal ID for terminal B

Detailed description

PARAM_L

Signature: *[static,const]* unsigned long **PARAM_L**

Description: A constant giving the parameter ID for parameter L

Python specific notes:

	The object exposes a readable attribute 'PARAM_L'. This is the getter.
TERMINAL_A	**Signature**: *[static,const]* unsigned long **TERMINAL_A** **Description**: A constant giving the terminal ID for terminal A **Python specific notes:** The object exposes a readable attribute 'TERMINAL_A'. This is the getter.
TERMINAL_B	**Signature**: *[static,const]* unsigned long **TERMINAL_B** **Description**: A constant giving the terminal ID for terminal B **Python specific notes:** The object exposes a readable attribute 'TERMINAL_B'. This is the getter.
_assign	**Signature**: void **_assign** (const DeviceClassInductor other) **Description**: Assigns another object to self
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_dup	**Signature**: *[const]* new DeviceClassInductor ptr **_dup** **Description**: Creates a copy of self
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

4.148. API reference - Class DeviceClassDiode

Notation used in Ruby API documentation

Module: db

Description: A device class for a diode.

Class hierarchy: DeviceClassDiode » DeviceClass

This class describes a diode. A diode has two terminals, A (anode) and C (cathode). It has two parameters: The diode area in square micrometers (A) and the diode area perimeter in micrometers (P).

Diodes only combine when parallel and in the same direction. In this case, their areas and perimeters are added. This class has been introduced in version 0.26.

Public methods

	void	**_assign**	(const DeviceClas other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new DeviceClassDiode ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

Public static methods and constants

[static,const]	unsigned long	**PARAM_A**	A constant giving the parameter ID for parameter A
[static,const]	unsigned long	**PARAM_P**	A constant giving the parameter ID for parameter P
[static,const]	unsigned long	**TERMINAL_A**	A constant giving the terminal ID for terminal A
[static,const]	unsigned long	**TERMINAL_C**	A constant giving the terminal ID for terminal C

Detailed description

PARAM_A

Signature: *[static,const]* unsigned long **PARAM_A**

Description: A constant giving the parameter ID for parameter A

Python specific notes:
The object exposes a readable attribute 'PARAM_A'. This is the getter.

PARAM_P

Signature: *[static,const]* unsigned long **PARAM_P**

Description: A constant giving the parameter ID for parameter P

Python specific notes:
The object exposes a readable attribute 'PARAM_P'. This is the getter.

TERMINAL_A

Signature: *[static,const]* unsigned long **TERMINAL_A**

Description: A constant giving the terminal ID for terminal A

Python specific notes:
The object exposes a readable attribute 'TERMINAL_A'. This is the getter.

TERMINAL_C

Signature: *[static,const]* unsigned long **TERMINAL_C**

Description: A constant giving the terminal ID for terminal C

Python specific notes:
The object exposes a readable attribute 'TERMINAL_C'. This is the getter.

_assign

Signature: void **_assign** (const DeviceClassDiode other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new DeviceClassDiode ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

4.149. API reference - Class DeviceClassBJT3Transistor

Notation used in Ruby API documentation

Module: db

Description: A device class for a bipolar transistor.

Class hierarchy: DeviceClassBJT3Transistor » DeviceClass

This class describes a bipolar transistor. Bipolar transistors have tree terminals: the collector (C), the base (B) and the emitter (E). Multi-emitter transistors are resolved in individual devices. The parameters are AE, AB and AC for the emitter, base and collector areas in square micrometers and PE, PB and PC for the emitter, base and collector perimeters in micrometers. In addition, the emitter count (NE) is given. The emitter count is 1 always for a transistor extracted initially. Upon combination of devices, the emitter counts are added, thus forming multi-emitter devices.

This class has been introduced in version 0.26.

Public methods

	void	**_assign**	(const DeviceClass other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new DeviceClassBJT3Transis ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

Public static methods and constants

[static,const]	unsigned long	**PARAM_AB**	A constant giving the parameter ID for parameter AB (base area)
[static,const]	unsigned long	**PARAM_AC**	A constant giving the parameter ID for parameter AC (collector area)
[static,const]	unsigned long	**PARAM_AE**	A constant giving the parameter ID for parameter AE (emitter area)
[static,const]	unsigned long	**PARAM_NE**	A constant giving the parameter ID for parameter NE (emitter count)
[static,const]	unsigned long	**PARAM_PB**	A constant giving the parameter ID for parameter PB (base perimeter)

[static,const]	unsigned long	**PARAM_PC**	A constant giving the parameter ID for parameter PC (collector perimeter)
[static,const]	unsigned long	**PARAM_PE**	A constant giving the parameter ID for parameter PE (emitter perimeter)
[static,const]	unsigned long	**TERMINAL_B**	A constant giving the terminal ID for terminal B (base)
[static,const]	unsigned long	**TERMINAL_C**	A constant giving the terminal ID for terminal C (collector)
[static,const]	unsigned long	**TERMINAL_E**	A constant giving the terminal ID for terminal E (emitter)

Detailed description

PARAM_AB

Signature: *[static,const]* unsigned long **PARAM_AB**

Description: A constant giving the parameter ID for parameter AB (base area)

Python specific notes:
The object exposes a readable attribute 'PARAM_AB'. This is the getter.

PARAM_AC

Signature: *[static,const]* unsigned long **PARAM_AC**

Description: A constant giving the parameter ID for parameter AC (collector area)

Python specific notes:
The object exposes a readable attribute 'PARAM_AC'. This is the getter.

PARAM_AE

Signature: *[static,const]* unsigned long **PARAM_AE**

Description: A constant giving the parameter ID for parameter AE (emitter area)

Python specific notes:
The object exposes a readable attribute 'PARAM_AE'. This is the getter.

PARAM_NE

Signature: *[static,const]* unsigned long **PARAM_NE**

Description: A constant giving the parameter ID for parameter NE (emitter count)

Python specific notes:
The object exposes a readable attribute 'PARAM_NE'. This is the getter.

PARAM_PB

Signature: *[static,const]* unsigned long **PARAM_PB**

Description: A constant giving the parameter ID for parameter PB (base perimeter)

Python specific notes:
The object exposes a readable attribute 'PARAM_PB'. This is the getter.

PARAM_PC

Signature: *[static,const]* unsigned long **PARAM_PC**

Description: A constant giving the parameter ID for parameter PC (collector perimeter)

Python specific notes:
The object exposes a readable attribute 'PARAM_PC'. This is the getter.

PARAM_PE

Signature: *[static,const]* unsigned long **PARAM_PE**

Description: A constant giving the parameter ID for parameter PE (emitter perimeter)

	Python specific notes: The object exposes a readable attribute 'PARAM_PE'. This is the getter.
TERMINAL_B	**Signature**: *[static,const]* unsigned long **TERMINAL_B** **Description**: A constant giving the terminal ID for terminal B (base) **Python specific notes:** The object exposes a readable attribute 'TERMINAL_B'. This is the getter.
TERMINAL_C	**Signature**: *[static,const]* unsigned long **TERMINAL_C** **Description**: A constant giving the terminal ID for terminal C (collector) **Python specific notes:** The object exposes a readable attribute 'TERMINAL_C'. This is the getter.
TERMINAL_E	**Signature**: *[static,const]* unsigned long **TERMINAL_E** **Description**: A constant giving the terminal ID for terminal E (emitter) **Python specific notes:** The object exposes a readable attribute 'TERMINAL_E'. This is the getter.
_assign	**Signature**: void **_assign** (const DeviceClassBJT3Transistor other) **Description**: Assigns another object to self
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_dup	**Signature**: *[const]* new DeviceClassBJT3Transistor ptr **_dup** **Description**: Creates a copy of self
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

4.150. API reference - Class DeviceClassBJT4Transistor

Notation used in Ruby API documentation

Module: db

Description: A device class for a 4-terminal bipolar transistor.

Class hierarchy: DeviceClassBJT4Transistor » DeviceClassBJT3Transistor » DeviceClass

This class describes a bipolar transistor with a substrate terminal. A device class for a bipolar transistor without a substrate terminal is DeviceClassBJT3Transistor. The additional terminal is 'S' for the substrate terminal. BJT4 transistors combine in parallel if both substrate terminals are connected to the same net.

This class has been introduced in version 0.26.

Public methods

	void	**_assign**	(const DeviceClass other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new DeviceClassBJT4Transis ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

Public static methods and constants

[static,const]	unsigned long	**TERMINAL_S**	A constant giving the terminal ID for terminal S

Detailed description

TERMINAL_S

Signature: *[static,const]* unsigned long **TERMINAL_S**

Description: A constant giving the terminal ID for terminal S

Python specific notes:
The object exposes a readable attribute 'TERMINAL_S'. This is the getter.

_assign

Signature: void **_assign** (const DeviceClassBJT4Transistor other)

Description: Assigns another object to self

_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_dup	**Signature**: *[const]* new DeviceClassBJT4Transistor ptr **_dup** **Description**: Creates a copy of self
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.

4.151. API reference - Class DeviceClassMOS3Transistor

Notation used in Ruby API documentation

Module: db

Description: A device class for a 3-terminal MOS transistor.

Class hierarchy: DeviceClassMOS3Transistor » DeviceClass

This class describes a MOS transistor without a bulk terminal. A device class for a MOS transistor with a bulk terminal is DeviceClassMOS4Transistor. MOS transistors are defined by their combination behavior and the basic parameters.

The parameters are L, W, AS, AD, PS and PD for the gate length and width in micrometers, source and drain area in square micrometers and the source and drain perimeter in micrometers.

The terminals are S, G and D for source, gate and drain.

MOS transistors combine in parallel mode, when both gate lengths are identical and their gates are connected (source and drain can be swapped). In this case, their widths and source and drain areas are added.

This class has been introduced in version 0.26.

Public methods

	void	**_assign**	(const DeviceClassM other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new DeviceClassMOS3Trar ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	void	**join_split_gates**	(Circuit ptr circuit)	Joins source/drain nets from 'split gate' transistor strings on the given circuit

Public static methods and constants

[static,const]	unsigned long	**PARAM_AD**	A constant giving the parameter ID for parameter AD
[static,const]	unsigned long	**PARAM_AS**	A constant giving the parameter ID for parameter AS
[static,const]	unsigned long	**PARAM_L**	A constant giving the parameter ID for parameter L

[static,const]	unsigned long	**PARAM_PD**	A constant giving the parameter ID for parameter PD
[static,const]	unsigned long	**PARAM_PS**	A constant giving the parameter ID for parameter PS
[static,const]	unsigned long	**PARAM_W**	A constant giving the parameter ID for parameter W
[static,const]	unsigned long	**TERMINAL_D**	A constant giving the terminal ID for terminal D
[static,const]	unsigned long	**TERMINAL_G**	A constant giving the terminal ID for terminal G
[static,const]	unsigned long	**TERMINAL_S**	A constant giving the terminal ID for terminal S

Detailed description

PARAM_AD

Signature: *[static,const]* unsigned long **PARAM_AD**

Description: A constant giving the parameter ID for parameter AD

Python specific notes:
The object exposes a readable attribute 'PARAM_AD'. This is the getter.

PARAM_AS

Signature: *[static,const]* unsigned long **PARAM_AS**

Description: A constant giving the parameter ID for parameter AS

Python specific notes:
The object exposes a readable attribute 'PARAM_AS'. This is the getter.

PARAM_L

Signature: *[static,const]* unsigned long **PARAM_L**

Description: A constant giving the parameter ID for parameter L

Python specific notes:
The object exposes a readable attribute 'PARAM_L'. This is the getter.

PARAM_PD

Signature: *[static,const]* unsigned long **PARAM_PD**

Description: A constant giving the parameter ID for parameter PD

Python specific notes:
The object exposes a readable attribute 'PARAM_PD'. This is the getter.

PARAM_PS

Signature: *[static,const]* unsigned long **PARAM_PS**

Description: A constant giving the parameter ID for parameter PS

Python specific notes:
The object exposes a readable attribute 'PARAM_PS'. This is the getter.

PARAM_W

Signature: *[static,const]* unsigned long **PARAM_W**

Description: A constant giving the parameter ID for parameter W

Python specific notes:
The object exposes a readable attribute 'PARAM_W'. This is the getter.

TERMINAL_D

Signature: *[static,const]* unsigned long **TERMINAL_D**

Description: A constant giving the terminal ID for terminal D

Python specific notes:
The object exposes a readable attribute 'TERMINAL_D'. This is the getter.

TERMINAL_G

Signature: *[static,const]* unsigned long **TERMINAL_G**

Description: A constant giving the terminal ID for terminal G

Python specific notes:
The object exposes a readable attribute 'TERMINAL_G'. This is the getter.

TERMINAL_S

Signature: *[static,const]* unsigned long **TERMINAL_S**

Description: A constant giving the terminal ID for terminal S

Python specific notes:
The object exposes a readable attribute 'TERMINAL_S'. This is the getter.

_assign

Signature: void **_assign** (const DeviceClassMOS3Transistor other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new DeviceClassMOS3Transistor ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

	Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
join_split_gates	**Signature**: *[const]* void **join_split_gates** (Circuit ptr circuit) **Description**: Joins source/drain nets from 'split gate' transistor strings on the given circuit This method has been introduced in version 0.27.9

4.152. API reference - Class DeviceClassMOS4Transistor

Notation used in Ruby API documentation

Module: db

Description: A device class for a 4-terminal MOS transistor.

Class hierarchy: DeviceClassMOS4Transistor » DeviceClassMOS3Transistor » DeviceClass

This class describes a MOS transistor with a bulk terminal. A device class for a MOS transistor without a bulk terminal is DeviceClassMOS3Transistor. MOS transistors are defined by their combination behavior and the basic parameters.

The additional terminal is 'B' for the bulk terminal. MOS4 transistors combine in parallel if both bulk terminals are connected to the same net.

This class has been introduced in version 0.26.

Public methods

	void	**_assign**	(const DeviceClass other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new DeviceClassMOS4Trans ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.

Public static methods and constants

[static,const]	unsigned long	**TERMINAL_B**	A constant giving the terminal ID for terminal B

Detailed description

TERMINAL_B

Signature: *[static,const]* unsigned long **TERMINAL_B**

Description: A constant giving the terminal ID for terminal B

Python specific notes:
The object exposes a readable attribute 'TERMINAL_B'. This is the getter.

_assign

Signature: void **_assign** (const DeviceClassMOS4Transistor other)

Description: Assigns another object to self

_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_dup	**Signature**: *[const]* new DeviceClassMOS4Transistor ptr **_dup** **Description**: Creates a copy of self
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.

4.153. API reference - Class DeviceClassFactory

Notation used in Ruby API documentation

Module: db

Description: A factory for creating specific device classes for the standard device extractors

Use a reimplementation of this class to provide a device class generator for built-in device extractors such as DeviceExtractorMOS3Transistor. The constructor of this extractor has a 'factory' parameter which takes an object of DeviceClassFactory type.

If such an object is provided, this factory is used to create the actual device class. The following code shows an example:

```
class MyClass < RBA::DeviceClassMOS3Transistor
  ... overrides some methods ...
end

class MyFactory < RBA::DeviceClassFactory
  def create_class
    MyClass.new
  end
end

extractor = RBA::DeviceExtractorMOS3Transistor::new("NMOS", false, MyFactory.new)
```

When using a factory with a device extractor, make sure it creates a corresponding device class, e.g. for the DeviceExtractorMOS3Transistor extractor create a device class derived from DeviceClassMOS3Transistor.

This class has been introduced in version 0.27.3.

Public constructors

new DeviceClassFactory ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const DeviceCla other)	Assigns another object to self
[virtual,const]	new DeviceClass ptr	**create_class**		Creates the DeviceClass object

[const]	new DeviceClassFactory ptr	**dup**	Creates a copy of self

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const DeviceClassFactory other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

create_class

Signature: *[virtual,const]* new DeviceClass ptr **create_class**

Description: Creates the DeviceClass object

Reimplement this method to create the desired device class.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new DeviceClassFactory ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new DeviceClassFactory ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

4.154. API reference - Class NetlistDeviceExtractorLayerDefinition

Notation used in Ruby API documentation

Module: db

Description: Describes a layer used in the device extraction

This read-only structure is used to describe a layer in the device extraction. Every device has specific layers used in the device extraction process. Layer definitions can be retrieved using NetlistDeviceExtractor#each_layer.

This class has been introduced in version 0.26.

Public constructors

new NetlistDeviceExtractorLayerDefinition ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetlistDevi other)	Assigns another object to self
[const]	string	**description**		Gets the description of the layer.
[const]	new NetlistDeviceExtractorLay ptr	**dup**		Creates a copy of self
[const]	unsigned long	**fallback_index**		Gets the index of the fallback layer.
[const]	unsigned long	**index**		Gets the index of the layer.
[const]	string	**name**		Gets the name of the layer.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead

[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const NetlistDeviceExtractorLayerDefinition other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

description

Signature: *[const]* string **description**

Description: Gets the description of the layer.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetlistDeviceExtractorLayerDefinition ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

fallback_index

Signature: *[const]* unsigned long **fallback_index**

Description: Gets the index of the fallback layer.

This is the index of the layer to be used when this layer isn't specified for input or (more important) output.

index

Signature: *[const]* unsigned long **index**

Description: Gets the index of the layer.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

name

Signature: *[const]* string **name**

Description: Gets the name of the layer.

new

Signature: *[static]* new NetlistDeviceExtractorLayerDefinition ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

4.155. API reference - Class DeviceExtractorBase

Notation used in Ruby API documentation

Module: db

Description: The base class for all device extractors.

This is an abstract base class for device extractors. See GenericDeviceExtractor for a generic class which you can reimplement to supply your own customized device extractor. In many cases using one of the preconfigured specific device extractors may be useful already and it's not required to implement a custom one. For an example about a preconfigured device extractor see DeviceExtractorMOS3Transistor.

This class cannot and should not be instantiated explicitly. Use one of the subclasses instead.

This class has been introduced in version 0.26.

Public constructors

new DeviceExtractorBase ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	DeviceClass ptr	**device_class**		Gets the device class used during extraction
[const,iter]	NetlistDeviceExtractorLayerDefinition	**each_layer_definition**		Iterates over all layer definitions.
[iter]	LogEntryData	**each_log_entry**		Iterates over all log entries collected in the device extractor.Starting with version 0.28.13, the preferred name of the method is 'each_log_entry' as log entries have been generalized to become warnings too.
[const]	string	**name**		Gets the name of the device extractor and the device class.
	void	**name=**	(string name)	Sets the name of the device extractor and the device class.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead

[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[iter]	LogEntryData	**each_error**	Use of this method is deprecated. Use each_log_entry instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

device_class

Signature: DeviceClass ptr **device_class**

Description: Gets the device class used during extraction

The attribute will hold the actual device class used in the device extraction. It is valid only after 'extract_devices'.

This method has been added in version 0.27.3.

each_error

Signature: *[iter]* LogEntryData **each_error**

Description: Iterates over all log entries collected in the device extractor.Starting with version 0.28.13, the preferred name of the method is 'each_log_entry' as log entries have been generalized to become warnings too.

Use of this method is deprecated. Use each_log_entry instead

each_layer_definition

Signature: *[const,iter]* NetlistDeviceExtractorLayerDefinition **each_layer_definition**

Description: Iterates over all layer definitions.

each_log_entry

Signature: *[iter]* LogEntryData **each_log_entry**

Description: Iterates over all log entries collected in the device extractor.Starting with version 0.28.13, the preferred name of the method is 'each_log_entry' as log entries have been generalized to become warnings too.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

	This method returns true, if self is a const reference. In that case, only const methods may be called on self.
name	**Signature**: *[const]* string **name** **Description**: Gets the name of the device extractor and the device class. **Python specific notes:** The object exposes a readable attribute 'name'. This is the getter.
name=	**Signature**: void **name=** (string name) **Description**: Sets the name of the device extractor and the device class. **Python specific notes:** The object exposes a writable attribute 'name'. This is the setter.
new	**Signature**: *[static]* new DeviceExtractorBase ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.

4.156. API reference - Class GenericDeviceExtractor

Notation used in Ruby API documentation

Module: db

Description: The basic class for implementing custom device extractors.

Class hierarchy: GenericDeviceExtractor » DeviceExtractorBase

This class serves as a base class for implementing customized device extractors. This class does not provide any extraction functionality, so you have to implement every detail.

Device extraction requires a few definitions. The definitions are made in the reimplementation of the setup method. Required definitions to be made are:

- The name of the extractor. This will also be the name of the device class produced by the extractor. The name is set using name=.
- The device class of the devices to produce. The device class is registered using register_device_class.
- The layers used for the device extraction. These are input layers for the extraction as well as output layers for defining the terminals. Terminals are the points at which the nets connect to the devices. Layers are defined using define_layer. Initially, layers are abstract definitions with a name and a description. Concrete layers will be given when defining the connectivity.

When the device extraction is started, the device extraction algorithm will first ask the device extractor for the 'connectivity'. This is not a connectivity in a sense of electrical connections. The connectivity defines are logical compound that makes up the device. 'Connected' shapes are collected and presented to the device extractor. The connectivity is obtained by calling get_connectivity. This method must be implemented to produce the connectivity.

Finally, the individual devices need to be extracted. Each cluster of connected shapes is presented to the device extractor. A cluster may include more than one device. It's the device extractor's responsibility to extract the devices from this cluster and deliver the devices through create_device. In addition, terminals have to be defined, so the net extractor can connect to the devices. Terminal definitions are made through define_terminal. The device extraction is implemented in the extract_devices method.

If errors occur during device extraction, the error method may be used to issue such errors. Errors reported this way are kept in the error log.

This class has been introduced in version 0.26.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	Device ptr	**create_device**		Creates a device.
[const]	double	**dbu**		Gets the database unit
	NetlistDeviceExtrac	**define_layer**	(string name, string description)	Defines a layer.

	NetlistDeviceExtractorLayerDefinition	**define_layer**	(string name, unsigned long fallback, string description)	Defines a layer with a fallback layer.
	void	**define_terminal**	(Device ptr device, unsigned long terminal_id, unsigned long layer_index, const Polygon shape)	Defines a device terminal.
	void	**define_terminal**	(Device ptr device, unsigned long terminal_id, unsigned long layer_index, const Box shape)	Defines a device terminal.
	void	**define_terminal**	(Device ptr device, unsigned long terminal_id, unsigned long layer_index, const Point point)	Defines a device terminal.
	void	**define_terminal**	(Device ptr device, string terminal_name, string layer_name, const Polygon shape)	Defines a device terminal using names for terminal and layer.
	void	**define_terminal**	(Device ptr device, string terminal_name, string layer_name, const Box shape)	Defines a device terminal using names for terminal and layer.
	void	**define_terminal**	(Device ptr device, string terminal_name, string layer_name, const Point point)	Defines a device terminal using names for terminal and layer.
	void	**error**	(string message)	Issues an error with the given message
	void	**error**	(string message, const DPolygon geometry)	Issues an error with the given message and micrometer-units polygon geometry
	void	**error**	(string message, const Polygon geometry)	Issues an error with the given message and database-unit polygon geometry
	void	**error**	(string category_name, string category_description, string message)	Issues an error with the given category name and description, message
	void	**error**	(string category_name, string category_description, string message, const DPolygon geometry)	Issues an error with the given category name and description, message and micrometer-units polygon geometry
	void	**error**	(string category_name, string category_description, string message, const Polygon geometry)	Issues an error with the given category name and description, message and database-unit polygon geometry
[virtual]	void	**extract_devices**	(Region[] layer_geometry)	Extracts the devices from the given shape cluster.

[virtual,const]	Connectivity	**get_connectivity**	(const Layout layout, unsigned int[] layers)	Gets the connectivity object used to extract the device geometry.
	void	**register_device_cla**	(DeviceClass ptr device_class)	Registers a device class.
[const]	double	**sdbu**		Gets the scaled database unit
[virtual]	void	**setup**		Sets up the extractor.
	void	**warn**	(string message)	Issues a warning with the given message
	void	**warn**	(string message, const DPolygon geometry)	Issues a warning with the given message and micrometer-units polygon geometry
	void	**warn**	(string message, const Polygon geometry)	Issues a warning with the given message and database-unit polygon geometry
	void	**warn**	(string category_name, string category_description, string message)	Issues a warning with the given category name and description, message
	void	**warn**	(string category_name, string category_description, string message, const DPolygon geometry)	Issues a warning with the given category name and description, message and micrometer-units polygon geometry
	void	**warn**	(string category_name, string category_description, string message, const Polygon geometry)	Issues a warning with the given category name and description, message and database-unit polygon geometry

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create_device

Signature: Device ptr **create_device**

Description: Creates a device.

The device object returned can be configured by the caller, e.g. set parameters. It will be owned by the netlist and must not be deleted by the caller.

dbu

Signature: *[const]* double **dbu**

Description: Gets the database unit

define_layer

Signature: NetlistDeviceExtractorLayerDefinition **define_layer** (string name, string description)

Description: Defines a layer.

Returns: The layer descriptor object created for this layer (use 'index' to get the layer's index)

Each call will define one more layer for the device extraction. This method shall be used inside the implementation of setup to define the device layers. The actual geometries are later available to extract_devices in the order the layers are defined.

define_opt_layer

Signature: NetlistDeviceExtractorLayerDefinition **define_opt_layer** (string name, unsigned long fallback, string description)

Description: Defines a layer with a fallback layer.

Returns: The layer descriptor object created for this layer (use 'index' to get the layer's index)

As define_layer, this method allows specification of device extraction layer. In addition to define_layout, it features a fallback layer. If in the device extraction statement, the primary layer is not given, the fallback layer will be used. Hence, this layer is optional. The fallback layer is given by its index and must be defined before the layer using the fallback layer is defined. For the index, 0 is the first layer defined, 1 the second and so forth.

define_terminal

(1) Signature: void **define_terminal** (Device ptr device, unsigned long terminal_id, unsigned long layer_index, const Polygon shape)

Description: Defines a device terminal.

This method will define a terminal to the given device and the given terminal ID. The terminal will be placed on the layer given by "layer_index". The layer index is the index of the layer during layer definition. The first layer is 0, the second layer 1 etc.

This version produces a terminal with a shape given by the polygon. Note that the polygon is specified in database units.

(2) Signature: void **define_terminal** (Device ptr device, unsigned long terminal_id, unsigned long layer_index, const Box shape)

Description: Defines a device terminal.

This method will define a terminal to the given device and the given terminal ID. The terminal will be placed on the layer given by "layer_index". The layer index is the index of the layer during layer definition. The first layer is 0, the second layer 1 etc.

This version produces a terminal with a shape given by the box. Note that the box is specified in database units.

(3) Signature: void **define_terminal** (Device ptr device, unsigned long terminal_id, unsigned long layer_index, const Point point)

Description: Defines a device terminal.

This method will define a terminal to the given device and the given terminal ID. The terminal will be placed on the layer given by "layer_index". The layer index is the index of the layer during layer definition. The first layer is 0, the second layer 1 etc.

This version produces a point-like terminal. Note that the point is specified in database units.

(4) Signature: void **define_terminal** (Device ptr device, string terminal_name, string layer_name, const Polygon shape)

Description: Defines a device terminal using names for terminal and layer.

This convenience version of the ID-based define_terminal methods allows using names for terminal and layer. It has been introduced in version 0.28.

(5) Signature: void **define_terminal** (Device ptr device, string terminal_name, string layer_name, const Box shape)

Description: Defines a device terminal using names for terminal and layer.

This convenience version of the ID-based define_terminal methods allows using names for terminal and layer. It has been introduced in version 0.28.

(6) Signature: void **define_terminal** (Device ptr device, string terminal_name, string layer_name, const Point point)

Description: Defines a device terminal using names for terminal and layer.

This convenience version of the ID-based define_terminal methods allows using names for terminal and layer. It has been introduced in version 0.28.

error

(1) Signature: void **error** (string message)

Description: Issues an error with the given message

(2) Signature: void **error** (string message, const DPolygon geometry)

Description: Issues an error with the given message and micrometer-units polygon geometry

(3) Signature: void **error** (string message, const Polygon geometry)

Description: Issues an error with the given message and database-unit polygon geometry

(4) Signature: void **error** (string category_name, string category_description, string message)

Description: Issues an error with the given category name and description, message

(5) Signature: void **error** (string category_name, string category_description, string message, const DPolygon geometry)

Description: Issues an error with the given category name and description, message and micrometer-units polygon geometry

(6) Signature: void **error** (string category_name, string category_description, string message, const Polygon geometry)

Description: Issues an error with the given category name and description, message and database-unit polygon geometry

extract_devices

Signature: *[virtual]* void **extract_devices** (Region[] layer_geometry)

Description: Extracts the devices from the given shape cluster.

The shape cluster is a set of geometries belonging together in terms of the connectivity defined by "get_connectivity". The cluster might cover multiple devices, so the implementation needs to consider this case. The geometries are already merged.

The implementation of this method shall use "create_device" to create new devices based on the geometry found. It shall use "define_terminal" to define terminals by which the nets extracted in the network extraction step connect to the new devices.

get_connectivity

Signature: *[virtual,const]* Connectivity **get_connectivity** (const Layout layout, unsigned int[] layers)

Description: Gets the connectivity object used to extract the device geometry.

This method shall raise an error, if the input layer are not properly defined (e.g. too few etc.)

This is not a connectivity definition in the electrical sense, but defines the cluster of shapes which generates a specific device. In this case, 'connectivity' means 'definition of shapes that need to touch to form the device'.

The 'layers' argument specifies the actual layer layouts for the logical device layers (see define_layer). The list of layers corresponds to the number of layers defined. Use the layer indexes from this list to build the connectivity with Connectivity#connect. Note, that in order to capture a connected cluster of shapes on the same layer you'll need to include a self-connection like 'connectivity.connect(layers[0], layers[0])'.

register_device_class

Signature: void **register_device_class** (DeviceClass ptr device_class)

Description: Registers a device class.

The device class object will become owned by the netlist and must not be deleted by the caller. The name of the device class will be changed to the name given to the device extractor. This method shall be used inside the implementation of setup to register the device classes.

sdbu

Signature: *[const]* double **sdbu**

Description: Gets the scaled database unit

Use this unit to compute device properties. It is the database unit multiplied with the device scaling factor.

setup

Signature: *[virtual]* void **setup**

Description: Sets up the extractor.

This method is supposed to set up the device extractor. This involves three basic steps: defining the name, the device class and setting up the device layers.

Use name= to give the extractor and its device class a name. Use register_device_class to register the device class you need. Defined the layers by calling define_layer once or several times.

warn

(1) Signature: void **warn** (string message)

Description: Issues a warning with the given message

Warnings have been introduced in version 0.28.13.

(2) Signature: void **warn** (string message, const DPolygon geometry)

Description: Issues a warning with the given message and micrometer-units polygon geometry

Warnings have been introduced in version 0.28.13.

(3) Signature: void **warn** (string message, const Polygon geometry)

Description: Issues a warning with the given message and database-unit polygon geometry

Warnings have been introduced in version 0.28.13.

(4) Signature: void **warn** (string category_name, string category_description, string message)

Description: Issues a warning with the given category name and description, message

Warnings have been introduced in version 0.28.13.

(5) Signature: void **warn** (string category_name, string category_description, string message, const DPolygon geometry)

Description: Issues a warning with the given category name and description, message and micrometer-units polygon geometry

Warnings have been introduced in version 0.28.13.

(6) Signature: void **warn** (string category_name, string category_description, string message, const Polygon geometry)

Description: Issues a warning with the given category name and description, message and database-unit polygon geometry

Warnings have been introduced in version 0.28.13.

4.157. API reference - Class DeviceExtractorMOS3Transistor

Notation used in Ruby API documentation

Module: db

Description: A device extractor for a three-terminal MOS transistor

Class hierarchy: DeviceExtractorMOS3Transistor » DeviceExtractorBase

This class supplies the generic extractor for a MOS device. The device is defined by two basic input layers: the diffusion area (source and drain) and the gate area. It requires a third layer (poly) to put the gate terminals on. The separation between poly and allows separating the device recognition layer (gate) from the conductive layer.

The device class produced by this extractor is DeviceClassMOS3Transistor.

The extractor delivers six parameters:

- 'L' - the gate length in micrometer units
- 'W' - the gate width in micrometer units
- 'AS' and 'AD' - the source and drain region areas in square micrometers
- 'PS' and 'PD' - the source and drain region perimeters in micrometer units

The device layer names are:

- In strict mode: 'S' (source), 'D' (drain) and 'G' (gate).
- In non-strict mode: 'SD' (source and drain) and 'G' (gate).

The terminals are output on these layers:

- 'tS' - source. Default output is 'S' (strict mode) or 'SD' (otherwise).
- 'tD' - drain. Default output is 'D' (strict mode) or 'SD' (otherwise).
- 'tG' - gate. Default output is 'G'.

The source/drain (diffusion) area is distributed on the number of gates connecting to the particular source or drain area.

This class is a closed one and methods cannot be reimplemented. To reimplement specific methods, see DeviceExtractor.

This class has been introduced in version 0.26.

Public constructors

new DeviceExtractorMOS3Transistor ptr	**new**	(string name, bool strict = false, DeviceClassFactory ptr factory = none)	Creates a new device extractor with the given name.

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed

[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.
[const]	bool	**strict?**	Returns a value indicating whether extraction happens in strict mode.

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

new

Signature: *[static]* new DeviceExtractorMOS3Transistor ptr **new** (string name, bool strict = false, DeviceClassFactory ptr factory = none)

Description: Creates a new device extractor with the given name.

If strict is true, the MOS device extraction will happen in strict mode. That is, source and drain are not interchangeable.

For the 'factory' parameter see DeviceClassFactory. It has been added in version 0.27.3.

Python specific notes:
This method is the default initializer of the object.

strict?

Signature: *[const]* bool **strict?**

Description: Returns a value indicating whether extraction happens in strict mode.

4.158. API reference - Class DeviceExtractorMOS4Transistor

Notation used in Ruby API documentation

Module: db

Description: A device extractor for a four-terminal MOS transistor

Class hierarchy: DeviceExtractorMOS4Transistor » DeviceExtractorBase

This class supplies the generic extractor for a MOS device. It is based on the DeviceExtractorMOS3Transistor class with the extension of a bulk terminal and corresponding bulk terminal output (annotation) layer.

The parameters of a MOS4 device are the same than for MOS3 devices. For the device layers the bulk layer is added.

- 'B' (bulk) - currently this layer is not used and can be empty.

The bulk terminals are output on this layer:

- 'tB' - bulk terminal (a copy of the gate shape). Default output is 'B'.

The bulk terminal layer can be empty. In this case, it needs to be connected to a global net to establish the net connection.

The device class produced by this extractor is DeviceClassMOS4Transistor.

This class is a closed one and methods cannot be reimplemented. To reimplement specific methods, see DeviceExtractor.

This class has been introduced in version 0.26.

Public constructors

new DeviceExtractorMOS4Transistor ptr	**new**	(string name, bool strict = false, DeviceClassFactory ptr factory = none)	Creates a new device extractor with the given name

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

new

Signature: *[static]* new DeviceExtractorMOS4Transistor ptr **new** (string name, bool strict = false, DeviceClassFactory ptr factory = none)

Description: Creates a new device extractor with the given name

For the 'factory' parameter see DeviceClassFactory. It has been added in version 0.27.3.

Python specific notes:
This method is the default initializer of the object.

4.159. API reference - Class DeviceExtractorResistor

Notation used in Ruby API documentation

Module: db

Description: A device extractor for a two-terminal resistor

Class hierarchy: DeviceExtractorResistor » DeviceExtractorBase

This class supplies the generic extractor for a resistor device. The device is defined by two geometry layers: the resistor 'wire' and two contacts per wire. The contacts should be attached to the ends of the wire. The wire length and width is computed from the edge lengths between the contacts and along the contacts respectively.

This simple computation is precise only when the resistor shape is a rectangle.

Using the given sheet resistance, the resistance value is computed by 'R = L / W * sheet_rho'.

The device class produced by this extractor is DeviceClassResistor. The extractor produces three parameters:

- 'R' - the resistance in Ohm
- 'A' - the resistor's area in square micrometer units
- 'P' - the resistor's perimeter in micrometer units

The device layer names are:

- 'R' - resistor path. This is the geometry that defines the resistor's current path.
- 'C' - contacts. These areas form the contact regions at the ends of the resistor path.

The terminals are output on these layers:

- 'tA', 'tB' - the two terminals of the resistor.

This class is a closed one and methods cannot be reimplemented. To reimplement specific methods, see DeviceExtractor.

This class has been introduced in version 0.26.

Public constructors

new DeviceExtractorResistor ptr	**new**	(string name, double sheet_rho, DeviceClassFactory ptr factory = none)	Creates a new device extractor with the given name

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

new

Signature: *[static]* new DeviceExtractorResistor ptr **new** (string name, double sheet_rho, DeviceClassFactory ptr factory = none)

Description: Creates a new device extractor with the given name

For the 'factory' parameter see DeviceClassFactory. It has been added in version 0.27.3.

Python specific notes:

This method is the default initializer of the object.

4.160. API reference - Class DeviceExtractorResistorWithBulk

Notation used in Ruby API documentation

Module: db

Description: A device extractor for a resistor with a bulk terminal

Class hierarchy: DeviceExtractorResistorWithBulk » DeviceExtractorBase

This class supplies the generic extractor for a resistor device including a bulk terminal. The device is defined the same way than devices are defined for DeviceExtractorResistor.

The device class produced by this extractor is DeviceClassResistorWithBulk. The extractor produces three parameters:

- 'R' - the resistance in Ohm
- 'A' - the resistor's area in square micrometer units
- 'P' - the resistor's perimeter in micrometer units

The device layer names are:

- 'R' - resistor path. This is the geometry that defines the resistor's current path.
- 'C' - contacts. These areas form the contact regions at the ends of the resistor path.
- 'W' - well, bulk. Currently this layer is ignored for the extraction and can be empty.

The terminals are output on these layers:

- 'tA', 'tB' - the two terminals of the resistor.
- 'tW' - the bulk terminal (copy of the resistor area).

The bulk terminal layer can be an empty layer representing the substrate. In this case, it needs to be connected globally.

This class is a closed one and methods cannot be reimplemented. To reimplement specific methods, see DeviceExtractor.

This class has been introduced in version 0.26.

Public constructors

new DeviceExtractorResistorWithBulk ptr	**new**	(string name, double sheet_rho, DeviceClassFactory ptr factory = none)	Creates a new device extractor with the given name

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

new

Signature: *[static]* new DeviceExtractorResistorWithBulk ptr **new** (string name, double sheet_rho, DeviceClassFactory ptr factory = none)

Description: Creates a new device extractor with the given name

For the 'factory' parameter see DeviceClassFactory. It has been added in version 0.27.3.

Python specific notes:

This method is the default initializer of the object.

4.161. API reference - Class DeviceExtractorCapacitor

Notation used in Ruby API documentation

Module: db

Description: A device extractor for a two-terminal capacitor

Class hierarchy: DeviceExtractorCapacitor » DeviceExtractorBase

This class supplies the generic extractor for a capacitor device. The device is defined by two geometry layers forming the 'plates' of the capacitor. The capacitance is computed from the overlapping area of the plates using 'C = A * area_cap' (area_cap is the capacitance per square micrometer area).

Although 'area_cap' can be given in any unit, Farad should be preferred as this is the convention used for output into a netlist.

The device class produced by this extractor is DeviceClassCapacitor. The extractor produces three parameters:

- 'C' - the capacitance
- 'A' - the capacitor's area in square micrometer units
- 'P' - the capacitor's perimeter in micrometer units

The device layer names are:

- 'P1', 'P2' - the two plates.

The terminals are output on these layers:

- 'tA', 'tB' - the two terminals. Defaults to 'P1' and 'P2'.

This class is a closed one and methods cannot be reimplemented. To reimplement specific methods, see DeviceExtractor.

This class has been introduced in version 0.26.

Public constructors

new DeviceExtractorCapacitor ptr	**new**	(string name, double area_cap, DeviceClassFactory ptr factory = none)	Creates a new device extractor with the given name

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

new

Signature: *[static]* new DeviceExtractorCapacitor ptr **new** (string name, double area_cap, DeviceClassFactory ptr factory = none)

Description: Creates a new device extractor with the given name

For the 'factory' parameter see DeviceClassFactory. It has been added in version 0.27.3.

Python specific notes:

This method is the default initializer of the object.

4.162. API reference - Class DeviceExtractorCapacitorWithBulk

Notation used in Ruby API documentation

Module: db

Description: A device extractor for a capacitor with a bulk terminal

Class hierarchy: DeviceExtractorCapacitorWithBulk » DeviceExtractorBase

This class supplies the generic extractor for a capacitor device including a bulk terminal. The device is defined the same way than devices are defined for DeviceExtractorCapacitor.

The device class produced by this extractor is DeviceClassCapacitorWithBulk. The extractor produces three parameters:

- 'C' - the capacitance
- 'A' - the capacitor's area in square micrometer units
- 'P' - the capacitor's perimeter in micrometer units

The device layer names are:

- 'P1', 'P2' - the two plates.
- 'W' - well, bulk. Currently this layer is ignored for the extraction and can be empty.

The terminals are output on these layers:

- 'tA', 'tB' - the two terminals. Defaults to 'P1' and 'P2'.
- 'tW' - the bulk terminal (copy of the resistor area).

The bulk terminal layer can be an empty layer representing the substrate. In this case, it needs to be connected globally.

This class is a closed one and methods cannot be reimplemented. To reimplement specific methods, see DeviceExtractor.

This class has been introduced in version 0.26.

Public constructors

new DeviceExtractorCapacitorWithBulk ptr	**new**	(string name, double sheet_rho, DeviceClassFactory ptr factory = none)	Creates a new device extractor with the given name

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

new

Signature: *[static]* new DeviceExtractorCapacitorWithBulk ptr **new** (string name, double sheet_rho, DeviceClassFactory ptr factory = none)

Description: Creates a new device extractor with the given name

For the 'factory' parameter see DeviceClassFactory. It has been added in version 0.27.3.

Python specific notes:

This method is the default initializer of the object.

4.163. API reference - Class DeviceExtractorBJT3Transistor

Notation used in Ruby API documentation

Module: db

Description: A device extractor for a bipolar transistor (BJT)

Class hierarchy: DeviceExtractorBJT3Transistor » DeviceExtractorBase

This class supplies the generic extractor for a bipolar transistor device.

Extraction of vertical and lateral transistors is supported through a generic geometry model: The basic area is the base area. A marker shape must be provided for this area. The emitter of the transistor is defined by emitter layer shapes inside the base area. Multiple emitter shapes can be present. In this case, multiple transistor devices sharing the same base and collector are generated. Finally, a collector layer can be given. If non-empty, the parts inside the base region will define the collector terminals. If empty, the collector is formed by the substrate. In this case, the base region will be output to the 'tC' terminal output layer. This layer then needs to be connected to a global net to form the net connection.

The device class produced by this extractor is DeviceClassBJT3Transistor. The extractor delivers these parameters:

- 'AE', 'AB' and 'AC' - the emitter, base and collector areas in square micrometer units
- 'PE', 'PB' and 'PC' - the emitter, base and collector perimeters in micrometer units
- 'NE' - emitter count (initially 1 but increases when devices are combined)

The device layer names are:

- 'E' - emitter.
- 'B' - base.
- 'C' - collector.

The terminals are output on these layers:

- 'tE' - emitter. Default output is 'E'.
- 'tB' - base. Default output is 'B'.
- 'tC' - collector. Default output is 'C'.

This class is a closed one and methods cannot be reimplemented. To reimplement specific methods, see DeviceExtractor.

This class has been introduced in version 0.26.

Public constructors

new DeviceExtractorBJT3Transistor ptr	**new**	(string name, DeviceClassFactory ptr factory = none)	Creates a new device extractor with the given name

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed

[const]	bool	_is_const_object?	Returns a value indicating whether the reference is a const reference
	void	_manage	Marks the object as managed by the script side.
	void	_unmanage	Marks the object as no longer owned by the script side.

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void _**manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void _**unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

new

Signature: *[static]* new DeviceExtractorBJT3Transistor ptr **new** (string name, DeviceClassFactory ptr factory = none)

Description: Creates a new device extractor with the given name

For the 'factory' parameter see DeviceClassFactory. It has been added in version 0.27.3.

Python specific notes:
This method is the default initializer of the object.

4.164. API reference - Class DeviceExtractorBJT4Transistor

Notation used in Ruby API documentation

Module: db

Description: A device extractor for a four-terminal bipolar transistor (BJT)

Class hierarchy: DeviceExtractorBJT4Transistor » DeviceExtractorBJT3Transistor » DeviceExtractorBase

This class supplies the generic extractor for a bipolar transistor device. It is based on the DeviceExtractorBJT3Transistor class with the extension of a substrate terminal and corresponding substrate terminal output (annotation) layer.

Two new layers are introduced:

- 'S' - the bulk (substrate) layer. Currently this layer is ignored and can be empty.
- 'tS' - the bulk terminal output layer (defaults to 'S').

The bulk terminal layer ('tS') can be an empty layer representing the wafer substrate. In this use mode the substrate terminal shapes will be produced on the 'tS' layer. This layer then needs to be connected to a global net to establish the net connection.

The device class produced by this extractor is DeviceClassBJT4Transistor. The This class is a closed one and methods cannot be reimplemented. To reimplement specific methods, see DeviceExtractor.

This class has been introduced in version 0.26.

Public constructors

new DeviceExtractorBJT4Transistor ptr	**new**	(string name, DeviceClassFactory ptr factory = none)	Creates a new device extractor with the given name

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Detailed description

_create

Signature: void _create

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

new

Signature: *[static]* new DeviceExtractorBJT4Transistor ptr **new** (string name, DeviceClassFactory ptr factory = none)

Description: Creates a new device extractor with the given name

For the 'factory' parameter see DeviceClassFactory. It has been added in version 0.27.3.

Python specific notes:
This method is the default initializer of the object.

4.165. API reference - Class DeviceExtractorDiode

Notation used in Ruby API documentation

Module: db

Description: A device extractor for a planar diode

Class hierarchy: DeviceExtractorDiode » DeviceExtractorBase

This class supplies the generic extractor for a planar diode. The diode is defined by two layers whose overlap area forms the diode. The p-type layer forms the anode, the n-type layer the cathode.

The device class produced by this extractor is DeviceClassDiode. The extractor extracts the two parameters of this class:

- 'A' - the diode area in square micrometer units.
- 'P' - the diode perimeter in micrometer units.

The device layers are:

- 'P' - the p doped area.
- 'N' - the n doped area.

The diode region is defined by the overlap of p and n regions.

The terminal output layers are:

- 'tA' - anode. Defaults to 'P'.
- 'tC' - cathode. Defaults to 'N'.

This class is a closed one and methods cannot be reimplemented. To reimplement specific methods, see DeviceExtractor.

This class has been introduced in version 0.26.

Public constructors

new DeviceExtractorDiode ptr	**new**	(string name, DeviceClassFactory ptr factory = none)	Creates a new device extractor with the given name

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

new

Signature: *[static]* new DeviceExtractorDiode ptr **new** (string name, DeviceClassFactory ptr factory = none)

Description: Creates a new device extractor with the given name

For the 'factory' parameter see DeviceClassFactory. It has been added in version 0.27.3.

Python specific notes:

This method is the default initializer of the object.

4.166. API reference - Class Connectivity

Notation used in Ruby API documentation

Module: db

Description: This class specifies connections between different layers.

Connections are build using connect. There are basically two flavours of connections: intra-layer and inter-layer.

Intra-layer connections make nets begin propagated along different shapes on the same net. Without the intra-layer connections, nets are not propagated over shape boundaries. As this is usually intended, intra-layer connections should always be specified for each layer.

Inter-layer connections connect shapes on different layers. Shapes which touch across layers will be connected if their layers are specified as being connected through inter-layer connect.

All layers are specified in terms of layer indexes. Layer indexes are layout layer indexes (see Layout class).

The connectivity object also manages the global nets. Global nets are substrate for example and they are propagated automatically from subcircuits to circuits. Global nets are defined by name and are managed through IDs. To get the name for a given ID, use global_net_name. Starting with version 0.29, soft connections are supported. Soft connections attach to high-ohmic substrate or diffusion layers (the 'lower' layer) are upon netlist extraction it will be checked that no wiring is routed over such connections. See soft_connect and soft_global_connect for details.

This class has been introduced in version 0.26.

Public constructors

new Connectivity ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Connectivity other)	Assigns another object to self
	void	**connect**	(unsigned int layer)	Specifies intra-layer connectivity.
	void	**connect**	(unsigned int layer_a, unsigned int layer_b)	Specifies inter-layer connectivity.
	unsigned long	**connect_global**	(unsigned int layer, string global_net_name)	Connects the given layer to the global net given by name.

[const]	new Connectivity ptr	**dup**		Creates a copy of self
	unsigned long	**global_net_id**	(string global_net_name)	Gets the ID for a given global net name.
[const]	string	**global_net_name**	(unsigned long global_net_id)	Gets the name for a given global net ID.
	void	**soft_connect**	(unsigned int layer_a, unsigned int layer_b)	Specifies a soft connection between layer_a and layer_b.
	unsigned long	**soft_connect_global**	(unsigned int layer, string global_net_name)	Soft-connects the given layer to the global net given by name.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Connectivity other)

Description: Assigns another object to self

connect

(1) Signature: void **connect** (unsigned int layer)

Description: Specifies intra-layer connectivity.

This method specifies a hard connection between shapes on the given layer. Without specifying such a connection, shapes on that layer do not form connection regions.

(2) Signature: void **connect** (unsigned int layer_a, unsigned int layer_b)

Description: Specifies inter-layer connectivity.

This method specifies a hard connection between shapes on layer_a and layer_b.

connect_global

Signature: unsigned long **connect_global** (unsigned int layer, string global_net_name)

Description: Connects the given layer to the global net given by name.

Returns the ID of the global net.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

	Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
dup	**Signature**: *[const]* new Connectivity ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
global_net_id	**Signature**: unsigned long **global_net_id** (string global_net_name) **Description**: Gets the ID for a given global net name.
global_net_name	**Signature**: *[const]* string **global_net_name** (unsigned long global_net_id) **Description**: Gets the name for a given global net ID.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**Signature**: *[static]* new Connectivity ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
soft_connect	**Signature**: void **soft_connect** (unsigned int layer_a, unsigned int layer_b) **Description**: Specifies a soft connection between layer_a and layer_b. **layer_a**: The 'upper' layer **layer_b**: The 'lower' layer Soft connections are made between a lower and an upper layer. The lower layer conceptually is a high-ohmic (i.e. substrate, diffusion) region that is not intended for signal wiring. The netlist extraction will check that no routing happens over such regions. Soft connections have in introduced in version 0.29.
soft_connect_global	**Signature**: unsigned long **soft_connect_global** (unsigned int layer, string global_net_name) **Description**: Soft-connects the given layer to the global net given by name. Returns the ID of the global net. See soft_connect for a description of the soft connection feature. The global net is always the 'lower' (i.e. high-ohmic, substrate) part of the soft connection.

Soft connections have in introduced in version 0.29.

4.167. API reference - Class LayoutToNetlist

Notation used in Ruby API documentation

Module: db

Description: A generic framework for extracting netlists from layouts

Sub-classes: BuildNetHierarchyMode

This class wraps various concepts from db::NetlistExtractor and db::NetlistDeviceExtractor and more. It is supposed to provide a framework for extracting a netlist from a layout.

The use model of this class consists of five steps which need to be executed in this order.

- Configuration: in this step, the LayoutToNetlist object is created and if required, configured. Methods to be used in this step are threads=, area_ratio= or max_vertex_count=. The constructor for the LayoutToNetlist object receives a RecursiveShapeIterator object which basically supplies the hierarchy and the layout taken as input.
- Preparation In this step, the device recognition and extraction layers are drawn from the framework. Derived can now be computed using boolean operations. Methods to use in this step are make_layer and its variants. Layer preparation is not necessarily required to happen before all other steps. Layers can be computed shortly before they are required.
- Following the preparation, the devices can be extracted using extract_devices. This method needs to be called for each device extractor required. Each time, a device extractor needs to be given plus a map of device layers. The device layers are device extractor specific. Either original or derived layers may be specified here. Layer preparation may happen between calls to extract_devices.
- Once the devices are derived, the netlist connectivity can be defined and the netlist extracted. The connectivity is defined with connect and its flavours. The actual netlist extraction happens with extract_netlist.
- After netlist extraction, the information is ready to be retrieved. The produced netlist is available with netlist. The Shapes of a specific net are available with shapes_of_net. probe_net allows finding a net by probing a specific location.

You can also use the extractor with an existing DeepShapeStore object or even flat data. In this case, preparation means importing existing regions with the register method. If you want to use the LayoutToNetlist object with flat data, use the 'LayoutToNetlist(topcell, dbu)' constructor. If you want to use it with hierarchical data and an existing DeepShapeStore object, use the 'LayoutToNetlist(dss)' constructor.

This class has been introduced in version 0.26.

Public constructors

new LayoutToNetlist ptr	**new**	(const RecursiveShapeIterator iter)	Creates a new extractor connected to an original layout
new LayoutToNetlist ptr	**new**		Creates a new and empty extractor object
new LayoutToNetlist ptr	**new**	(DeepShapeStore ptr dss)	Creates a new extractor object reusing an existing DeepShapeStore object
new LayoutToNetlist ptr	**new**	(DeepShapeStore ptr dss, unsigned int layout_index)	Creates a new extractor object reusing an existing DeepShapeStore object
new LayoutToNetlist ptr	**new**	(string topcell_name, double dbu)	Creates a new extractor object with a flat DSS

Public methods

void	**_create**	Ensures the C++ object is created
void	**_destroy**	Explicitly destroys the object

[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
	Region	antenna_check	(const Region gate, const Region metal, double ratio, variant[] diodes = [], Texts ptr texts = nil)	Runs an antenna check on the extracted clusters
	Region	antenna_check	(const Region gate, double gate_perimeter_factor, const Region metal, double metal_perimeter_factor, double ratio, variant[] diodes = [], Texts ptr texts = nil)	Runs an antenna check on the extracted clusters taking the perimeter into account
	Region	antenna_check	(const Region gate, double gate_area_factor, double gate_perimeter_factor, const Region metal, double metal_area_factor, double metal_perimeter_factor, double ratio, variant[] diodes = [], Texts ptr texts = nil)	Runs an antenna check on the extracted clusters taking the perimeter into account and providing an area factor
[const]	double	area_ratio		Gets the area_ratio parameter for the hierarchical network processor
	void	area_ratio=	(double r)	Sets the area_ratio parameter for the hierarchical network processor
[const]	void	build_all_nets	(const CellMapping cmap, Layout target, map<unsigned int,const Region ptr> lmap, variant net_cell_name_prefix = nil, variant netname_prop = nil, LayoutToNetlist::BuildNetHierarchyMode hier_mode = BNH_Flatten, variant circuit_cell_name_prefix = nil,	Builds a full hierarchical representation of the nets

			variant device_cell_name_prefix = nil)	
[const]	void	**build_net**	(const Net net, Layout target, Cell target_cell, map<unsigned int,const Region ptr> lmap, variant netname_prop = nil, LayoutToNetlist::BuildNetHier hier_mode = BNH_Flatten, variant circuit_cell_name_prefix = nil, variant device_cell_name_prefix = nil)	Builds a net representation in the given layout and cell
[const]	void	**build_nets**	(const Net ptr[] nets, const CellMapping cmap, Layout target, map<unsigned int,const Region ptr> lmap, variant net_cell_name_prefix = nil, variant netname_prop = nil, LayoutToNetlist::BuildNetHierarchyMode hier_mode = BNH_Flatten, variant circuit_cell_name_prefix = nil, variant device_cell_name_prefix = nil)	Like build_all_nets, but with the ability to select some nets.
	CellMapping	**cell_mapping_into**	(Layout layout, Cell cell, bool with_device_cells = false)	Creates a cell mapping for copying shapes from the internal layout to the given target layout.
	CellMapping	**cell_mapping_into**	(Layout layout, Cell cell, const Net ptr[] nets, bool with_device_cells = false)	Creates a cell mapping for copying shapes from the internal layout to the given target layout.
	void	**check_extraction_erro**		Raises an exception if extraction errors are present
	void	**clear_join_net_names**		Clears all implicit net joining expressions.
	void	**clear_join_nets**		Clears all explicit net joining expressions.
	void	**connect**	(const Region l)	Defines an intra-layer connection for the given layer.

	void	**connect**	(const Region a, const Region b)	Defines an inter-layer connection for the given layers.
	void	**connect**	(const Region a, const Texts b)	Defines an inter-layer connection for the given layers.
	void	**connect**	(const Texts a, const Region b)	Defines an inter-layer connection for the given layers.
	unsigned long	**connect_global**	(const Region l, string global_net_name)	Defines a connection of the given layer with a global net.
	unsigned long	**connect_global**	(const Texts l, string global_net_name)	Defines a connection of the given text layer with a global net.
	CellMapping	**const_cell_mapping_into**	(const Layout layout, const Cell cell)	Creates a cell mapping for copying shapes from the internal layout to the given target layout.
[const]	string	**description**		Gets the description of the database
	void	**description=**	(string description)	Sets the description of the database
[const]	double	**device_scaling**		Gets the device scaling factor
	void	**device_scaling=**	(double f)	Sets the device scaling factor
	DeepShapeStore	**dss**		Gets a reference to the internal DSS object.
[const,iter]	LogEntryData	**each_log_entry**		Iterates over all log entries collected during device and netlist extraction.
	void	**extract_devices**	(DeviceExtractorBase extractor, map<string,ShapeCollection ptr> layers)	Extracts devices
	void	**extract_netlist**		Runs the netlist extraction
[const]	string	**filename**		Gets the file name of the database
[const]	string	**generator**		Gets the generator string.
	void	**generator=**	(string generator)	Sets the generator string.
[const]	string	**global_net_name**	(unsigned long global_net_id)	Gets the global net name for the given global net ID.
[const]	bool	**include_floating_subc**		Gets a flag indicating whether to include floating subcircuits in the netlist.
	void	**include_floating_subcircuits=**	(bool flag)	Sets a flag indicating whether to include floating subcircuits in the netlist.
	Layout ptr	**internal_layout**		Gets the internal layout
	Cell ptr	**internal_top_cell**		Gets the internal top cell

[const]	bool	**is_extracted?**		Gets a value indicating whether the netlist has been extracted
[const]	bool	**is_persisted?**	(const Region layer)	Returns true, if the given layer is a persisted region.
[const]	bool	**is_persisted?**	(const Texts layer)	Returns true, if the given layer is a persisted texts collection.
	void	**join_net_names**	(string pattern)	Specifies another pattern for implicit joining of nets for the top level cell.
	void	**join_net_names**	(string cell_pattern, string pattern)	Specifies another pattern for implicit joining of nets for the cells from the given cell pattern.
	void	**join_nets**	(string[] net_names)	Specifies another name list for explicit joining of nets for the top level cell.
	void	**join_nets**	(string cell_pattern, string[] net_names)	Specifies another name list for explicit joining of nets for the cells from the given cell pattern.
	void	**keep_dss**		Resumes ownership over the DSS object if created with an external one.
	new Region ptr	**layer_by_index**	(unsigned int index)	Gets a layer object for the given index.
	new Region ptr	**layer_by_name**	(string name)	Gets a layer object for the given name.
[const]	string	**layer_name**	(const ShapeCollection l)	Gets the name of the given layer
[const]	string	**layer_name**	(unsigned int l)	Gets the name of the given layer (by index)
[const]	string[]	**layer_names**		Returns a list of names of the layer kept inside the LayoutToNetlist object.
[const]	unsigned int	**layer_of**	(const Region l)	Gets the internal layer for a given extraction layer
[const]	unsigned int	**layer_of**	(const Texts l)	Gets the internal layer for a given text collection
	new Region ptr	**make_layer**	(string name =)	Creates a new, empty hierarchical region
	new Region ptr	**make_layer**	(unsigned int layer_index, string name =)	Creates a new hierarchical region representing an original layer
	new Region ptr	**make_polygon_layer**	(unsigned int layer_index, string name =)	Creates a new region representing an original layer taking polygons and texts
	new Texts ptr	**make_text_layer**	(unsigned int layer_index, string name =)	Creates a new region representing an original layer taking texts only
[const]	unsigned long	**max_vertex_count**		
	void	**max_vertex_count=**	(unsigned long n)	Sets the max_vertex_count parameter for the hierarchical network processor

[const]	string	**name**		Gets the name of the database
	void	**name=**	(string name)	Sets the name of the database
[const]	Netlist ptr	**netlist**		gets the netlist extracted (0 if no extraction happened yet)
[const]	string	**original_file**		Gets the original file name of the database
	void	**original_file=**	(string path)	Sets the original file name of the database
	Net ptr	**probe_net**	(const Region of_layer, const DPoint point, SubCircuit ptr[] ptr sc_path_out = nil, Circuit ptr initial_circuit = nil)	Finds the net by probing a specific location on the given layer
	Net ptr	**probe_net**	(const Region of_layer, const Point point, SubCircuit ptr[] ptr sc_path_out = nil, Circuit ptr initial_circuit = nil)	Finds the net by probing a specific location on the given layer
	void	**read**	(string path)	Reads the extracted netlist from the file.
	void	**read_l2n**	(string path)	Reads the extracted netlist from the file.
	void	**register**	(const ShapeCollection l, string n =)	Names the given layer
	void	**reset_extracted**		Resets the extracted netlist and enables re-extraction
[const]	new Region ptr	**shapes_of_net**	(const Net net, const Region of_layer, bool recursive = true, const ICplxTrans trans = unity)	Returns all shapes of a specific net and layer.
[const]	void	**shapes_of_net**	(const Net net, const Region of_layer, bool recursive, Shapes to, unsigned long propid = 0, const ICplxTrans trans = unity)	Sends all shapes of a specific net and layer to the given Shapes container.
	void	**soft_connect**	(const Region a, const Region b)	Defines an inter-layer connection for the given layers in soft mode.
	void	**soft_connect**	(const Region a, const Texts b)	Defines an inter-layer connection for the given layers in soft mode.
	void	**soft_connect**	(const Texts a, const Region b)	Defines an inter-layer connection for the given layers in soft mode.

	unsigned long	**soft_connect_global**	(const Region l, string global_net_name)	Defines a connection of the given layer with a global net in soft mode.
	unsigned long	**soft_connect_global**	(const Texts l, string global_net_name)	Defines a connection of the given text layer with a global net in soft mode.
[const]	int	**threads**		Gets the number of threads to use for operations which support multiple threads
	void	**threads=**	(int n)	Sets the number of threads to use for operations which support multiple threads
[const]	bool	**top_level_mode**		Gets a flag indicating whether top level mode is enabled.
	void	**top_level_mode=**	(bool flag)	Sets a flag indicating whether top level mode is enabled.
	void	**write**	(string path, bool short_format = false)	Writes the extracted netlist to a file.
	void	**write_l2n**	(string path, bool short_format = false)	Writes the extracted netlist to a file.

Public static methods and constants

[static,const]	LayoutToNetlist::BuildNetHiera	**BNH_Disconnected**	This constant tells build_net and build_all_nets to produce local nets without connections to subcircuits (used for the "hier_mode" parameter).
[static,const]	LayoutToNetlist::BuildNetHierarchyMode	**BNH_Flatten**	This constant tells build_net and build_all_nets to flatten the nets (used for the "hier_mode" parameter).
[static,const]	LayoutToNetlist::BuildNetHiera	**BNH_SubcircuitCell**	This constant tells build_net and build_all_nets to produce a hierarchy of subcircuit cells per net (used for the "hier_mode" parameter).

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const,iter]	LogEntryData	**each_error**	Use of this method is deprecated. Use each_log_entry instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

BNH_Disconnected

Signature: *[static,const]* LayoutToNetlist::BuildNetHierarchyMode **BNH_Disconnected**

Description: This constant tells build_net and build_all_nets to produce local nets without connections to subcircuits (used for the "hier_mode" parameter).

Python specific notes:
The object exposes a readable attribute 'BNH_Disconnected'. This is the getter.

BNH_Flatten

Signature: *[static,const]* LayoutToNetlist::BuildNetHierarchyMode **BNH_Flatten**

Description: This constant tells build_net and build_all_nets to flatten the nets (used for the "hier_mode" parameter).

Python specific notes:
The object exposes a readable attribute 'BNH_Flatten'. This is the getter.

BNH_SubcircuitCells

Signature: *[static,const]* LayoutToNetlist::BuildNetHierarchyMode **BNH_SubcircuitCells**

Description: This constant tells build_net and build_all_nets to produce a hierarchy of subcircuit cells per net (used for the "hier_mode" parameter).

Python specific notes:
The object exposes a readable attribute 'BNH_SubcircuitCells'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is

known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void _**unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

antenna_check

(1) Signature: Region **antenna_check** (const Region gate, const Region metal, double ratio, variant[] diodes = [], Texts ptr texts = nil)

Description: Runs an antenna check on the extracted clusters

The antenna check will traverse all clusters and run an antenna check for all root clusters. The antenna ratio is defined by the total area of all "metal" shapes divided by the total area of all "gate" shapes on the cluster. Of all clusters where the antenna ratio is larger than the limit ratio all metal shapes are copied to the output region as error markers.

The simple call is:

```
l2n = ... # a LayoutToNetlist object
l2n.extract_netlist
# check for antenna ratio 10.0 of metal vs. poly:
errors = l2n.antenna(poly, metal, 10.0)
```

You can include diodes which rectify the antenna effect. Provide recognition layers for theses diodes and include them in the connections. Then specify the diode layers in the antenna call:

```
...
# include diode_layer1:
errors = l2n.antenna(poly, metal, 10.0, [ diode_layer1 ])
# include diode_layer1 and diode_layer2:errors = l2n.antenna(poly, metal,
 10.0, [ diode_layer1, diode_layer2 ])
```

Diodes can be configured to partially reduce the antenna effect depending on their area. This will make the diode_layer1 increase the ratio by 50.0 per square micrometer area of the diode:

```
...
# diode_layer1 increases the ratio by 50 per square micrometer area:
errors = l2n.antenna(poly, metal, 10.0 [ [ diode_layer, 50.0 ] ])
```

If 'texts' is non-nil, this text collection will receive labels explaining the error in terms of area values and relevant ratio.

The 'texts' parameter has been added in version 0.27.11.

(2) Signature: Region **antenna_check** (const Region gate, double gate_perimeter_factor, const Region metal, double metal_perimeter_factor, double ratio, variant[] diodes = [], Texts ptr texts = nil)

Description: Runs an antenna check on the extracted clusters taking the perimeter into account

This version of the antenna_check method allows taking the perimeter of gate or metal into account. The effective area is computed using:

```
Aeff = A + P * t
```

Here Aeff is the area used in the check, A is the polygon area, P the perimeter and t the perimeter factor. This formula applies to gate polygon area/perimeter with 'gate_perimeter_factor' for t and metal polygon area/perimeter with 'metal_perimeter_factor'. The perimeter_factor has the dimension of micrometers and can be thought of as the width of the material. Essentially the side walls of the material are taking into account for the surface area as well.

This variant has been introduced in version 0.26.6.

(3) Signature: Region **antenna_check** (const Region gate, double gate_area_factor, double gate_perimeter_factor, const Region metal, double metal_area_factor, double metal_perimeter_factor, double ratio, variant[] diodes = [], Texts ptr texts = nil)

Description: Runs an antenna check on the extracted clusters taking the perimeter into account and providing an area factor

This (most generic) version of the antenna_check method allows taking the perimeter of gate or metal into account and also provides a scaling factor for the area part. The effective area is computed using:

```
Aeff = A * f + P * t
```

Here f is the area factor and t the perimeter factor. A is the polygon area and P the polygon perimeter. A use case for this variant is to set the area factor to zero. This way, only perimeter contributions are considered.

This variant has been introduced in version 0.26.6.

area_ratio

Signature: *[const]* double **area_ratio**

Description: Gets the area_ratio parameter for the hierarchical network processor

See area_ratio= for details about this attribute.

Python specific notes:
The object exposes a readable attribute 'area_ratio'. This is the getter.

area_ratio=

Signature: void **area_ratio=** (double r)

Description: Sets the area_ratio parameter for the hierarchical network processor

This parameter controls splitting of large polygons in order to reduce the error made by the bounding box approximation.

Python specific notes:
The object exposes a writable attribute 'area_ratio'. This is the setter.

build_all_nets

Signature: *[const]* void **build_all_nets** (const CellMapping cmap, Layout target, map<unsigned int,const Region ptr> lmap, variant net_cell_name_prefix = nil, variant netname_prop = nil, LayoutToNetlist::BuildNetHierarchyMode hier_mode = BNH_Flatten, variant circuit_cell_name_prefix = nil, variant device_cell_name_prefix = nil)

Description: Builds a full hierarchical representation of the nets

cmap:	The mapping of internal layout to target layout for the circuit mapping
target:	The target layout
lmap:	Target layer indexes (keys) and net regions (values)
hier_mode:	See description of this method
netname_prop:	An (optional) property name to which to attach the net name
circuit_cell_name_prefix:	See method description
net_cell_name_prefix:	See method description
device_cell_name_prefix:	See above

This method copies all nets into cells corresponding to the circuits. It uses the 'cmap' object to determine the target cell (create it with "cell_mapping_into" or "const_cell_mapping_into"). If no mapping is provided for a specific circuit cell, the nets are copied into the next mapped parent as many times as the circuit cell appears there (circuit flattening).

The method has three net annotation modes:

- No annotation (net_cell_name_prefix == nil and netname_prop == nil): the shapes will be put into the target cell simply.
- Net name property (net_cell_name_prefix == nil and netname_prop != nil): the shapes will be annotated with a property named with netname_prop and containing the net name string.
- Individual subcells per net (net_cell_name_prefix != 0): for each net, a subcell is created and the net shapes will be put there (name of the subcell = net_cell_name_prefix + net name). (this mode can be combined with netname_prop too).

In addition, net hierarchy is covered in three ways:

- No connection indicated (hier_mode == BNH_Disconnected: the net shapes are simply put into their respective circuits. The connections are not indicated.
- Subnet hierarchy (hier_mode == BNH_SubcircuitCells): for each root net, a full hierarchy is built to accommodate the subnets (see build_net in recursive mode).
- Flat (hier_mode == BNH_Flatten): each net is flattened and put into the circuit it belongs to.

If a device cell name prefix is given, cells will be produced for each device abstract using a name like device_cell_name_prefix + device name. Otherwise the device shapes are treated as part of the net.

build_net

Signature: *[const]* void **build_net** (const Net net, Layout target, Cell target_cell, map<unsigned int,const Region ptr> lmap, variant netname_prop = nil, LayoutToNetlist::BuildNetHierarchyMode hier_mode = BNH_Flatten, variant circuit_cell_name_prefix = nil, variant device_cell_name_prefix = nil)

Description: Builds a net representation in the given layout and cell

target:	The target layout
target_cell:	The target cell
lmap:	Target layer indexes (keys) and net regions (values)
hier_mode:	See description of this method
netname_prop:	An (optional) property name to which to attach the net name
cell_name_prefix:	Chooses recursive mode if non-null
device_cell_name_prefix:	See above

This method puts the shapes of a net into the given target cell using a variety of options to represent the net name and the hierarchy of the net.

If the netname_prop name is not nil, a property with the given name is created and assigned the net name.

Net hierarchy is covered in three ways:

- No connection indicated (hier_mode == BNH_Disconnected: the net shapes are simply put into their respective circuits. The connections are not indicated.
- Subnet hierarchy (hier_mode == BNH_SubcircuitCells): for each root net, a full hierarchy is built to accommodate the subnets (see build_net in recursive mode).
- Flat (hier_mode == BNH_Flatten): each net is flattened and put into the circuit it belongs to.

If a device cell name prefix is given, cells will be produced for each device abstract using a name like device_cell_name_prefix + device name. Otherwise the device shapes are treated as part of the net.

build_nets

Signature: *[const]* void **build_nets** (const Net ptr[] nets, const CellMapping cmap, Layout target, map<unsigned int,const Region ptr> lmap, variant net_cell_name_prefix = nil, variant netname_prop = nil, LayoutToNetlist::BuildNetHierarchyMode hier_mode = BNH_Flatten, variant circuit_cell_name_prefix = nil, variant device_cell_name_prefix = nil)

Description: Like build_all_nets, but with the ability to select some nets.

cell_mapping_into

(1) Signature: CellMapping **cell_mapping_into** (Layout layout, Cell cell, bool with_device_cells = false)

Description: Creates a cell mapping for copying shapes from the internal layout to the given target layout.

If 'with_device_cells' is true, cells will be produced for devices. These are cells not corresponding to circuits, so they are disabled normally. Use this option, if you want to access device terminal shapes per device.

CAUTION: this function may create new cells in 'layout'. Use const_cell_mapping_into if you want to use the target layout's hierarchy and not modify it.

(2) Signature: CellMapping **cell_mapping_into** (Layout layout, Cell cell, const Net ptr[] nets, bool with_device_cells = false)

Description: Creates a cell mapping for copying shapes from the internal layout to the given target layout.

This version will only create cells which are required to represent the nets from the 'nets' argument.

If 'with_device_cells' is true, cells will be produced for devices. These are cells not corresponding to circuits, so they are disabled normally. Use this option, if you want to access device terminal shapes per device.

CAUTION: this function may create new cells in 'layout'. Use const_cell_mapping_into if you want to use the target layout's hierarchy and not modify it.

check_extraction_errors

Signature: void **check_extraction_errors**

Description: Raises an exception if extraction errors are present

This method has been introduced in version 0.28.13.

clear_join_net_names

Signature: void **clear_join_net_names**

Description: Clears all implicit net joining expressions.

See extract_netlist for more details about this feature.

This method has been introduced in version 0.27 and replaces the arguments of extract_netlist.

clear_join_nets

Signature: void **clear_join_nets**

Description: Clears all explicit net joining expressions.

See extract_netlist for more details about this feature.

Explicit net joining has been introduced in version 0.27.

connect

(1) Signature: void **connect** (const Region l)

Description: Defines an intra-layer connection for the given layer.

The layer is either an original layer created with make_includelayer and its variants or a derived layer. Certain limitations apply. It's safe to use boolean operations for deriving layers. Other operations are applicable as long as they are capable of delivering hierarchical layers.

(2) Signature: void **connect** (const Region a, const Region b)

Description: Defines an inter-layer connection for the given layers.

The conditions mentioned with intra-layer connect apply for this method too.

(3) Signature: void **connect** (const Region a, const Texts b)

Description: Defines an inter-layer connection for the given layers.

The conditions mentioned with intra-layer connect apply for this method too. As one argument is a (hierarchical) text collection, this method is used to attach net labels to polygons.

This variant has been introduced in version 0.27.

(4) Signature: void **connect** (const Texts a, const Region b)

Description: Defines an inter-layer connection for the given layers.

The conditions mentioned with intra-layer connect apply for this method too. As one argument is a (hierarchical) text collection, this method is used to attach net labels to polygons.

This variant has been introduced in version 0.27.

connect_global

(1) Signature: unsigned long **connect_global** (const Region l, string global_net_name)

Description: Defines a connection of the given layer with a global net.

This method returns the ID of the global net. Use global_net_name to get the name back from the ID.

(2) Signature: unsigned long **connect_global** (const Texts l, string global_net_name)

Description: Defines a connection of the given text layer with a global net.

This method returns the ID of the global net. Use global_net_name to get the name back from the ID.

This variant has been introduced in version 0.27.

const_cell_mapping_into

Signature: CellMapping **const_cell_mapping_into** (const Layout layout, const Cell cell)

Description: Creates a cell mapping for copying shapes from the internal layout to the given target layout.

This version will not create new cells in the target layout. If the required cells do not exist there yet, flatting will happen.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

description

Signature: *[const]* string **description**

Description: Gets the description of the database

Python specific notes:
The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string description)

Description: Sets the description of the database

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

device_scaling

Signature: *[const]* double **device_scaling**

Description: Gets the device scaling factor

See device_scaling= for details about this attribute.

Python specific notes:
The object exposes a readable attribute 'device_scaling'. This is the getter.

device_scaling=

Signature: void **device_scaling=** (double f)

Description: Sets the device scaling factor

This factor will scale the physical properties of the extracted devices accordingly. The scale factor applies an isotropic shrink (<1) or expansion (>1).

Python specific notes:
The object exposes a writable attribute 'device_scaling'. This is the setter.

dss

Signature: DeepShapeStore **dss**

Description: Gets a reference to the internal DSS object.

each_error

Signature: *[const,iter]* LogEntryData **each_error**

Description: Iterates over all log entries collected during device and netlist extraction.

Use of this method is deprecated. Use each_log_entry instead

This method has been introduced in version 0.28.13.

each_log_entry

Signature: *[const,iter]* LogEntryData **each_log_entry**

Description: Iterates over all log entries collected during device and netlist extraction.

This method has been introduced in version 0.28.13.

extract_devices

Signature: void **extract_devices** (DeviceExtractorBase extractor, map<string,ShapeCollection ptr> layers)

Description: Extracts devices

See the class description for more details. This method will run device extraction for the given extractor. The layer map is specific for the extractor and uses the region objects derived with make_layer and its variants.

In addition, derived regions can be passed too. Certain limitations apply. It's safe to use boolean operations for deriving layers. Other operations are applicable as long as they are capable of delivering hierarchical layers.

If errors occur, the device extractor will contain theses errors.

extract_netlist

Signature: void **extract_netlist**

Description: Runs the netlist extraction

See the class description for more details.

This method has been made parameter-less in version 0.27. Use include_floating_subcircuits= and join_net_names as substitutes for the arguments of previous versions.

filename

Signature: *[const]* string **filename**

Description: Gets the file name of the database

The filename is the name under which the database is stored or empty if it is not associated with a file.

generator

Signature: *[const]* string **generator**

Description: Gets the generator string.

The generator is the script that created this database.

Python specific notes:
The object exposes a readable attribute 'generator'. This is the getter.

generator=

Signature: void **generator=** (string generator)

Description: Sets the generator string.

Python specific notes:
The object exposes a writable attribute 'generator'. This is the setter.

global_net_name

Signature: *[const]* string **global_net_name** (unsigned long global_net_id)

Description: Gets the global net name for the given global net ID.

include_floating_subcircuits

Signature: *[const]* bool **include_floating_subcircuits**

Description: Gets a flag indicating whether to include floating subcircuits in the netlist.

See include_floating_subcircuits= for details.

This attribute has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'include_floating_subcircuits'. This is the getter.

include_floating_subcircuits=

Signature: void **include_floating_subcircuits=** (bool flag)

Description: Sets a flag indicating whether to include floating subcircuits in the netlist.

With 'include_floating_subcircuits' set to true, subcircuits with no connection to their parent circuit are still included in the circuit as floating subcircuits. Specifically on flattening this means that these subcircuits are properly propagated to their parent instead of appearing as additional top circuits.

This attribute has been introduced in version 0.27 and replaces the arguments of extract_netlist.

Python specific notes:
The object exposes a writable attribute 'include_floating_subcircuits'. This is the setter.

internal_layout

Signature: Layout ptr **internal_layout**

Description: Gets the internal layout

Usually it should not be required to obtain the internal layout. If you need to do so, make sure not to modify the layout as the functionality of the netlist extractor depends on it.

internal_top_cell

Signature: Cell ptr **internal_top_cell**

Description: Gets the internal top cell

Usually it should not be required to obtain the internal cell. If you need to do so, make sure not to modify the cell as the functionality of the netlist extractor depends on it.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_extracted?

Signature: *[const]* bool **is_extracted?**

Description: Gets a value indicating whether the netlist has been extracted

This method has been introduced in version 0.27.1.

is_persisted?

(1) Signature: *[const]* bool **is_persisted?** (const Region layer)

Description: Returns true, if the given layer is a persisted region.

Persisted layers are kept inside the LayoutToNetlist object and are not released if their object is destroyed. Named layers are persisted, unnamed layers are not. Only persisted, named layers can be put into connect.

(2) Signature: *[const]* bool **is_persisted?** (const Texts layer)

Description: Returns true, if the given layer is a persisted texts collection.

Persisted layers are kept inside the LayoutToNetlist object and are not released if their object is destroyed. Named layers are persisted, unnamed layers are not. Only persisted, named layers can be put into connect.

The variant for Texts collections has been added in version 0.27.

join_net_names

(1) Signature: void **join_net_names** (string pattern)

Description: Specifies another pattern for implicit joining of nets for the top level cell.

Use this method to register a pattern for net labels considered in implicit net joining. Implicit net joining allows connecting multiple parts of the same nets (e.g. supply rails) without need for a physical connection. The pattern specifies labels to look for. When parts are labelled with a name matching the expression, the parts carrying the same name are joined.

This method adds a new pattern. Use clear_join_net_names to clear the registered pattern.

Each pattern is a glob expression. Valid glob expressions are:

- "" no implicit connections.
- "*" to make all labels candidates for implicit connections.
- "VDD" to make all 'VDD" nets candidates for implicit connections.
- "VDD" to make all 'VDD'+suffix nets candidates for implicit connections.
- "{VDD,VSS}" to all VDD and VSS nets candidates for implicit connections.

Label matching is case sensitive.

This method has been introduced in version 0.27 and replaces the arguments of extract_netlist.

(2) Signature: void **join_net_names** (string cell_pattern, string pattern)

Description: Specifies another pattern for implicit joining of nets for the cells from the given cell pattern.

This method allows applying implicit net joining for specific cells, not only for the top cell.

This method adds a new pattern. Use clear_join_net_names to clear the registered pattern.

This method has been introduced in version 0.27 and replaces the arguments of extract_netlist.

join_nets

(1) Signature: void **join_nets** (string[] net_names)

Description: Specifies another name list for explicit joining of nets for the top level cell.

Use this method to join nets from the set of net names. All these nets will be connected together forming a single net. Explicit joining will imply implicit joining for the involved nets - partial nets involved will be connected too (intra-net joining).

This method adds a new name list. Use clear_join_nets to clear the registered pattern.

Explicit net joining has been introduced in version 0.27.

(2) Signature: void **join_nets** (string cell_pattern, string[] net_names)

Description: Specifies another name list for explicit joining of nets for the cells from the given cell pattern.

This method allows applying explicit net joining for specific cells, not only for the top cell.

This method adds a new name list. Use clear_join_nets to clear the registered pattern.

	Explicit net joining has been introduced in version 0.27.
keep_dss	**Signature**: void **keep_dss** **Description**: Resumes ownership over the DSS object if created with an external one.
layer_by_index	**Signature**: new Region ptr **layer_by_index** (unsigned int index) **Description**: Gets a layer object for the given index. Only named layers can be retrieved with this method. The returned object is a copy which represents the named layer.
layer_by_name	**Signature**: new Region ptr **layer_by_name** (string name) **Description**: Gets a layer object for the given name. The returned object is a copy which represents the named layer.
layer_name	**(1) Signature**: *[const]* string **layer_name** (const ShapeCollection l) **Description**: Gets the name of the given layer **(2) Signature**: *[const]* string **layer_name** (unsigned int l) **Description**: Gets the name of the given layer (by index)
layer_names	**Signature**: *[const]* string[] **layer_names** **Description**: Returns a list of names of the layer kept inside the LayoutToNetlist object.
layer_of	**(1) Signature**: *[const]* unsigned int **layer_of** (const Region l) **Description**: Gets the internal layer for a given extraction layer This method is required to derive the internal layer index - for example for investigating the cluster tree. **(2) Signature**: *[const]* unsigned int **layer_of** (const Texts l) **Description**: Gets the internal layer for a given text collection This method is required to derive the internal layer index - for example for investigating the cluster tree. The variant for Texts collections has been added in version 0.27.
make_layer	**(1) Signature**: new Region ptr **make_layer** (string name =) **Description**: Creates a new, empty hierarchical region The name is optional. If given, the layer will already be named accordingly (see register). **(2) Signature**: new Region ptr **make_layer** (unsigned int layer_index, string name =) **Description**: Creates a new hierarchical region representing an original layer 'layer_index' is the layer index of the desired layer in the original layout. This variant produces polygons and takes texts for net name annotation. A variant not taking texts is make_polygon_layer. A Variant only taking texts is make_text_layer. The name is optional. If given, the layer will already be named accordingly (see register).

make_polygon_layer

Signature: new Region ptr **make_polygon_layer** (unsigned int layer_index, string name =)

Description: Creates a new region representing an original layer taking polygons and texts

See make_layer for details.

The name is optional. If given, the layer will already be named accordingly (see register).

make_text_layer

Signature: new Texts ptr **make_text_layer** (unsigned int layer_index, string name =)

Description: Creates a new region representing an original layer taking texts only

See make_layer for details.

The name is optional. If given, the layer will already be named accordingly (see register).

Starting with version 0.27, this method returns a Texts object.

max_vertex_count

Signature: *[const]* unsigned long **max_vertex_count**

Description:

See max_vertex_count= for details about this attribute.

Python specific notes:
The object exposes a readable attribute 'max_vertex_count'. This is the getter.

max_vertex_count=

Signature: void **max_vertex_count=** (unsigned long n)

Description: Sets the max_vertex_count parameter for the hierarchical network processor

This parameter controls splitting of large polygons in order to enhance performance for very big polygons.

Python specific notes:
The object exposes a writable attribute 'max_vertex_count'. This is the setter.

name

Signature: *[const]* string **name**

Description: Gets the name of the database

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string name)

Description: Sets the name of the database

Python specific notes:
The object exposes a writable attribute 'name'. This is the setter.

netlist

Signature: *[const]* Netlist ptr **netlist**

Description: gets the netlist extracted (0 if no extraction happened yet)

new

(1) Signature: *[static]* new LayoutToNetlist ptr **new** (const RecursiveShapeIterator iter)

Description: Creates a new extractor connected to an original layout

This constructor will attach the extractor to an original layout through the shape iterator.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new LayoutToNetlist ptr **new**

Description: Creates a new and empty extractor object

The main objective for this constructor is to create an object suitable for reading an annotated netlist.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new LayoutToNetlist ptr **new** (DeepShapeStore ptr dss)

Description: Creates a new extractor object reusing an existing DeepShapeStore object

This constructor can be used if there is a DSS object already from which the shapes can be taken. This version can only be used with register to add layers (regions) inside the 'dss' object.

The make_... methods will not create new layers as there is no particular place defined where to create the layers.

The extractor will not take ownership of the dss object unless you call keep_dss.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new LayoutToNetlist ptr **new** (DeepShapeStore ptr dss, unsigned int layout_index)

Description: Creates a new extractor object reusing an existing DeepShapeStore object

This constructor can be used if there is a DSS object already from which the shapes can be taken. NOTE: in this case, the make_... functions will create new layers inside this DSS. To register existing layers (regions) use register.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new LayoutToNetlist ptr **new** (string topcell_name, double dbu)

Description: Creates a new extractor object with a flat DSS

topcell_name:	The name of the top cell of the internal flat layout
dbu:	The database unit to use for the internal flat layout

This constructor will create an extractor for flat extraction. Layers registered with register will be flattened. New layers created with make_... will be flat layers.

The database unit is mandatory because the physical parameter extraction for devices requires this unit for translation of layout to physical dimensions.

Python specific notes:
This method is the default initializer of the object.

original_file

Signature: *[const]* string **original_file**

Description: Gets the original file name of the database

The original filename is the layout file from which the netlist DB was created.

Python specific notes:
The object exposes a readable attribute 'original_file'. This is the getter.

original_file=

Signature: void **original_file=** (string path)

Description: Sets the original file name of the database

Python specific notes:
The object exposes a writable attribute 'original_file'. This is the setter.

probe_net

(1) Signature: Net ptr **probe_net** (const Region of_layer, const DPoint point, SubCircuit ptr[] ptr sc_path_out = nil, Circuit ptr initial_circuit = nil)

Description: Finds the net by probing a specific location on the given layer

This method will find a net looking at the given layer at the specific position. It will traverse the hierarchy below if no shape in the requested layer is found in the specified location. The function will report the topmost net from far above the hierarchy of circuits as possible.

If initial_circuit is given, the probing will start from this circuit and from the cell this circuit represents. By default, the probing will start from the top circuit.

If no net is found at all, 0 is returned.

It is recommended to use probe_net on the netlist right after extraction. Optimization functions such as Netlist#purge will remove parts of the net which means shape to net probing may no longer work for these nets.

If non-null and an array, 'sc_path_out' will receive a list of SubCircuits objects which lead to the net from the top circuit of the database.

This variant accepts a micrometer-unit location. The location is given in the coordinate space of the initial cell.

The sc_path_out and initial_circuit parameters have been added in version 0.27.

(2) Signature: Net ptr **probe_net** (const Region of_layer, const Point point, SubCircuit ptr[] ptr sc_path_out = nil, Circuit ptr initial_circuit = nil)

Description: Finds the net by probing a specific location on the given layer

See the description of the other probe_net variant. This variant accepts a database-unit location. The location is given in the coordinate space of the initial cell.

The sc_path_out and initial_circuit parameters have been added in version 0.27.

read

Signature: void **read** (string path)

Description: Reads the extracted netlist from the file.

This method employs the native format of KLayout.

read_l2n

Signature: void **read_l2n** (string path)

Description: Reads the extracted netlist from the file.

This method employs the native format of KLayout.

register

Signature: void **register** (const ShapeCollection l, string n =)

Description: Names the given layer

'l' must be a Region or Texts object. Flat regions or text collections must be registered with this function, before they can be used in connect. Registering will copy the shapes into the LayoutToNetlist object in this step to enable netlist extraction.

Naming a layer allows the system to indicate the layer in various contexts, i.e. when writing the data to a file. Named layers are also persisted inside the LayoutToNetlist object. They are not discarded when the Region object is destroyed.

If required, the system will assign a name automatically. This method has been generalized in version 0.27.

reset_extracted

Signature: void **reset_extracted**

Description: Resets the extracted netlist and enables re-extraction

This method is implicitly called when using connect or connect_global after a netlist has been extracted. This enables incremental connect with re-extraction.

This method has been introduced in version 0.27.1.

shapes_of_net

(1) Signature: *[const]* new Region ptr **shapes_of_net** (const Net net, const Region of_layer, bool recursive = true, const ICplxTrans trans = unity)

Description: Returns all shapes of a specific net and layer.

If 'recursive" is true, the returned region will contain the shapes of all subcircuits too.

The optional 'trans' parameter allows applying a transformation to all shapes. It has been introduced in version 0.28.4.

(2) Signature: *[const]* void **shapes_of_net** (const Net net, const Region of_layer, bool recursive, Shapes to, unsigned long propid = 0, const ICplxTrans trans = unity)

Description: Sends all shapes of a specific net and layer to the given Shapes container.

If 'recursive" is true, the returned region will contain the shapes of all subcircuits too. "prop_id" is an optional properties ID. If given, this property set will be attached to the shapes. The optional 'trans' parameter allows applying a transformation to all shapes. It has been introduced in version 0.28.4.

soft_connect

(1) Signature: void **soft_connect** (const Region a, const Region b)

Description: Defines an inter-layer connection for the given layers in soft mode.

Connects two layers through a soft connection. Soft connections cannot make connections between two different nets. These are directional connections where 'b' is the 'lower' layer (typically high-ohmic substrate or diffusion).

Soft connections have been introduced in version 0.29.

(2) Signature: void **soft_connect** (const Region a, const Texts b)

Description: Defines an inter-layer connection for the given layers in soft mode.

Connects two layers through a soft connection. Soft connections cannot make connections between two different nets. These are directional connections where 'b' is the 'lower' layer (typically high-ohmic substrate or diffusion). As one argument is a (hierarchical) text collection, this method is used to attach net labels to polygons.

Soft connections have been introduced in version 0.29.

(3) Signature: void **soft_connect** (const Texts a, const Region b)

Description: Defines an inter-layer connection for the given layers in soft mode.

Connects two layers through a soft connection. Soft connections cannot make connections between two different nets. These are directional connections where 'b' is the 'lower' layer (typically high-ohmic substrate or diffusion). As one argument is a (hierarchical) text collection, this method is used to attach net labels to polygons.

Soft connections have been introduced in version 0.29.

soft_connect_global

(1) Signature: unsigned long **soft_connect_global** (const Region l, string global_net_name)

Description: Defines a connection of the given layer with a global net in soft mode.

This method returns the ID of the global net. Use global_net_name to get the name back from the ID. Soft connections are directional, where the global net is the 'lower' layer (typically high-ohmic substrate or diffusion).

Soft connections have been introduced in version 0.29.

(2) Signature: unsigned long **soft_connect_global** (const Texts l, string global_net_name)

Description: Defines a connection of the given text layer with a global net in soft mode.

This method returns the ID of the global net. Use global_net_name to get the name back from the ID. Soft connections are directional, where the global net is the 'lower' layer (typically high-ohmic substrate or diffusion).

Soft connections have been introduced in version 0.29.

threads

Signature: *[const]* int **threads**

Description: Gets the number of threads to use for operations which support multiple threads

Python specific notes:
The object exposes a readable attribute 'threads'. This is the getter.

threads=

Signature: void **threads=** (int n)

Description: Sets the number of threads to use for operations which support multiple threads

Python specific notes:
The object exposes a writable attribute 'threads'. This is the setter.

top_level_mode

Signature: *[const]* bool **top_level_mode**

Description: Gets a flag indicating whether top level mode is enabled.

See top_level_mode= for details.

This attribute has been introduced in version 0.28.13.

Python specific notes:
The object exposes a readable attribute 'top_level_mode'. This is the getter.

top_level_mode=

Signature: void **top_level_mode=** (bool flag)

Description: Sets a flag indicating whether top level mode is enabled.

In top level mode, must-connect warnings are turned into errors for example. To enable top level mode, set this attribute to true. By default, top-level mode is turned off.

This attribute has been introduced in version 0.28.13.

Python specific notes:
The object exposes a writable attribute 'top_level_mode'. This is the setter.

write

Signature: void **write** (string path, bool short_format = false)

Description: Writes the extracted netlist to a file.

This method employs the native format of KLayout.

write_l2n

Signature: void **write_l2n** (string path, bool short_format = false)

Description: Writes the extracted netlist to a file.

This method employs the native format of KLayout.

4.168. API reference - Class LayoutToNetlist::BuildNetHierarchyMode

Notation used in Ruby API documentation

Module: db

Description: This class represents the LayoutToNetlist::BuildNetHierarchyMode enum

This class is equivalent to the class LayoutToNetlist::BuildNetHierarchyMode

This enum is used for LayoutToNetlist#build_all_nets and LayoutToNetlist#build_net.

Public constructors

new LayoutToNetlist::BuildNetHierarchyMode ptr	**new**	(int i)	Creates an enum from an integer value
new LayoutToNetlist::BuildNetHierarchyMode ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const LayoutToNetlist::BuildNetH other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const LayoutToNetlist::BuildNetH other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const LayoutToNetlist::BuildNetH other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	LayoutToNetlist::BuildNetHiera	**BNH_Disconnected**	This constant tells build_net and build_all_nets to produce local nets without connections to subcircuits (used for the "hier_mode" parameter).

[static,const]	LayoutToNetlist::BuildNetHierarchyMode **BNH_Flatten**	This constant tells build_net and build_all_nets to flatten the nets (used for the "hier_mode" parameter).
[static,const]	LayoutToNetlist::BuildNetHierarchyMode **BNH_SubcircuitCell**	This constant tells build_net and build_all_nets to produce a hierarchy of subcircuit cells per net (used for the "hier_mode" parameter).

Detailed description

!=

(1) Signature: *[const]* bool **!=** (const LayoutToNetlist::BuildNetHierarchyMode other)

Description: Compares two enums for inequality

(2) Signature: *[const]* bool **!=** (int other)

Description: Compares an enum with an integer for inequality

<

(1) Signature: *[const]* bool **<** (const LayoutToNetlist::BuildNetHierarchyMode other)

Description: Returns true if the first enum is less (in the enum symbol order) than the second

(2) Signature: *[const]* bool **<** (int other)

Description: Returns true if the enum is less (in the enum symbol order) than the integer value

==

(1) Signature: *[const]* bool **==** (const LayoutToNetlist::BuildNetHierarchyMode other)

Description: Compares two enums

(2) Signature: *[const]* bool **==** (int other)

Description: Compares an enum with an integer value

BNH_Disconnected

Signature: *[static,const]* LayoutToNetlist::BuildNetHierarchyMode **BNH_Disconnected**

Description: This constant tells build_net and build_all_nets to produce local nets without connections to subcircuits (used for the "hier_mode" parameter).

Python specific notes:
The object exposes a readable attribute 'BNH_Disconnected'. This is the getter.

BNH_Flatten

Signature: *[static,const]* LayoutToNetlist::BuildNetHierarchyMode **BNH_Flatten**

Description: This constant tells build_net and build_all_nets to flatten the nets (used for the "hier_mode" parameter).

Python specific notes:
The object exposes a readable attribute 'BNH_Flatten'. This is the getter.

BNH_SubcircuitCells

Signature: *[static,const]* LayoutToNetlist::BuildNetHierarchyMode **BNH_SubcircuitCells**

Description: This constant tells build_net and build_all_nets to produce a hierarchy of subcircuit cells per net (used for the "hier_mode" parameter).

Python specific notes:
The object exposes a readable attribute 'BNH_SubcircuitCells'. This is the getter.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

new

(1) Signature: *[static]* new LayoutToNetlist::BuildNetHierarchyMode ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new LayoutToNetlist::BuildNetHierarchyMode ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.169. API reference - Class DeepShapeStore

Notation used in Ruby API documentation

Module: db

Description: An opaque layout heap for the deep region processor

This class is used for keeping intermediate, hierarchical data for the deep region processor. It is used in conjunction with the region constructor to create a deep (hierarchical) region.

```
layout = ... # a layout
layer = ...  # a layer
cell = ...   # a cell (initial cell for the deep region)
dss = RBA::DeepShapeStore::new
region = RBA::Region::new(cell.begin(layer), dss)
```

The DeepShapeStore object also supplies some configuration options for the operations acting on the deep regions. See for example threads=.

This class has been introduced in version 0.26.

Public constructors

new DeepShapeStore ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add_breakout_cell**	(unsigned int layout_index, unsigned int[] cell_index)	Adds a cell indexe to the breakout cell list for the given layout inside the store
	void	**add_breakout_cells**	(unsigned int layout_index, unsigned int[] cells)	Adds cell indexes to the breakout cell list for the given layout inside the store
	void	**add_breakout_cells**	(unsigned int layout_index, string pattern)	Adds cells (given by a cell name pattern) to the breakout cell list for the given layout inside the store

	void	add_breakout_cells	(string pattern)	Adds cells (given by a cell name pattern) to the breakout cell list to all layouts inside the store
	void	clear_breakout_cells	(unsigned int layout_index)	Clears the breakout cells
	void	clear_breakout_cells		Clears the breakout cells
[const]	bool	is_singular?		Gets a value indicating whether there is a single layout variant
[const]	double	max_area_ratio		Gets the max. area ratio.
	void	max_area_ratio=	(double ratio)	Sets the max. area ratio for bounding box vs. polygon area
[const]	unsigned long	max_vertex_count		Gets the maximum vertex count.
	void	max_vertex_count=	(unsigned long count)	Sets the maximum vertex count default value
	void	pop_state		Restores the store's state on the state state
	void	push_state		Pushes the store's state on the state state
[const]	bool	reject_odd_polygons		Gets a flag indicating whether to reject odd polygons.
	void	reject_odd_polygons=	(bool count)	Sets a flag indicating whether to reject odd polygons
	void	set_breakout_cells	(unsigned int layout_index, unsigned int[] cells)	Sets the breakout cell list (as cell indexes) for the given layout inside the store
	void	set_breakout_cells	(unsigned int layout_index, string pattern)	Sets the breakout cell list (as cell name pattern) for the given layout inside the store
	void	set_breakout_cells	(string pattern)	Sets the breakout cell list (as cell name pattern) for the all layouts inside the store
	void	subcircuit_hierarchy_for_n	(bool value)	Sets a value indicating whether to build a subcircuit hierarchy per net
	void	subcircuit_hierarchy_for_nets=	(bool value)	Gets a value indicating whether to build a subcircuit hierarchy per net
[const]	int	text_enlargement		Gets the text enlargement value.
	void	text_enlargement=	(int value)	Sets the text enlargement value
[const]	variant	text_property_name		Gets the text property name.
	void	text_property_name=	(variant name)	Sets the text property name.
[const]	int	threads		Gets the number of threads.

	void	threads=	(int n)	Sets the number of threads to allocate for the hierarchical processor
[const]	bool	wants_all_cells		Gets a flag wether to copy the full hierarchy for the working layouts
	void	wants_all_cells=	(bool flag)	Sets a flag wether to copy the full hierarchy for the working layouts

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is

known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_breakout_cell

Signature: void **add_breakout_cell** (unsigned int layout_index, unsigned int[] cell_index)

Description: Adds a cell indexe to the breakout cell list for the given layout inside the store

See clear_breakout_cells for an explanation of breakout cells.

This method has been added in version 0.26.1

add_breakout_cells

(1) Signature: void **add_breakout_cells** (unsigned int layout_index, unsigned int[] cells)

Description: Adds cell indexes to the breakout cell list for the given layout inside the store

See clear_breakout_cells for an explanation of breakout cells.

This method has been added in version 0.26.1

(2) Signature: void **add_breakout_cells** (unsigned int layout_index, string pattern)

Description: Adds cells (given by a cell name pattern) to the breakout cell list for the given layout inside the store

See clear_breakout_cells for an explanation of breakout cells.

This method has been added in version 0.26.1

(3) Signature: void **add_breakout_cells** (string pattern)

Description: Adds cells (given by a cell name pattern) to the breakout cell list to all layouts inside the store

See clear_breakout_cells for an explanation of breakout cells.

This method has been added in version 0.26.1

clear_breakout_cells

(1) Signature: void **clear_breakout_cells** (unsigned int layout_index)

Description: Clears the breakout cells

Breakout cells are a feature by which hierarchy handling can be disabled for specific cells. If cells are specified as breakout cells, they don't interact with neighbor or parent cells, hence are virtually isolated. Breakout cells are useful to shortcut hierarchy evaluation for cells which are otherwise difficult to handle. An example are memory array cells with overlaps to their neighbors: a precise handling of such cells would generate variants and the boundary of the array. Although precise, this behavior leads to partial flattening and propagation of shapes. In consequence, this will also result in wrong device detection in LVS applications. In such cases, these array cells can be declared 'breakout cells' which makes them isolated entities and variant generation does not happen.

See also set_breakout_cells and add_breakout_cells.

	This method has been added in version 0.26.1 **(2) Signature**: void **clear_breakout_cells** **Description**: Clears the breakout cells See the other variant of clear_breakout_cells for details. This method has been added in version 0.26.1
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
is_singular?	**Signature**: *[const]* bool **is_singular?** **Description**: Gets a value indicating whether there is a single layout variant Specifically for network extraction, singular DSS objects are required. Multiple layouts may be present if different sources of layouts have been used. Such DSS objects are not usable for network extraction.
max_area_ratio	**Signature**: *[const]* double **max_area_ratio** **Description**: Gets the max. area ratio. **Python specific notes:** The object exposes a readable attribute 'max_area_ratio'. This is the getter.
max_area_ratio=	**Signature**: void **max_area_ratio=** (double ratio) **Description**: Sets the max. area ratio for bounding box vs. polygon area

This parameter is used to simplify complex polygons. It is used by create_polygon_layer with the default parameters. It's also used by boolean operations when they deliver their output.

Python specific notes:
The object exposes a writable attribute 'max_area_ratio'. This is the setter.

max_vertex_count

Signature: *[const]* unsigned long **max_vertex_count**

Description: Gets the maximum vertex count.

Python specific notes:
The object exposes a readable attribute 'max_vertex_count'. This is the getter.

max_vertex_count=

Signature: void **max_vertex_count=** (unsigned long count)

Description: Sets the maximum vertex count default value

This parameter is used to simplify complex polygons. It is used by create_polygon_layer with the default parameters. It's also used by boolean operations when they deliver their output.

Python specific notes:
The object exposes a writable attribute 'max_vertex_count'. This is the setter.

new

Signature: *[static]* new DeepShapeStore ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

pop_state

Signature: void **pop_state**

Description: Restores the store's state on the state state

This will restore the state pushed by push_state.

This method has been added in version 0.26.1

push_state

Signature: void **push_state**

Description: Pushes the store's state on the state state

This will save the stores state (threads, max_vertex_count, max_area_ratio, breakout cells ...) on the state stack. pop_state can be used to restore the state.

This method has been added in version 0.26.1

reject_odd_polygons

Signature: *[const]* bool **reject_odd_polygons**

Description: Gets a flag indicating whether to reject odd polygons.

This attribute has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'reject_odd_polygons'. This is the getter.

reject_odd_polygons=

Signature: void **reject_odd_polygons=** (bool count)

Description: Sets a flag indicating whether to reject odd polygons

Some kind of 'odd' (e.g. non-orientable) polygons may spoil the functionality because they cannot be handled properly. By using this flag, the shape store we reject these kind of polygons. The default is 'accept' (without warning).

This attribute has been introduced in version 0.27.

Python specific notes:

The object exposes a writable attribute 'reject_odd_polygons'. This is the setter.

set_breakout_cells

(1) Signature: void **set_breakout_cells** (unsigned int layout_index, unsigned int[] cells)

Description: Sets the breakout cell list (as cell indexes) for the given layout inside the store

See clear_breakout_cells for an explanation of breakout cells.

This method has been added in version 0.26.1

(2) Signature: void **set_breakout_cells** (unsigned int layout_index, string pattern)

Description: Sets the breakout cell list (as cell name pattern) for the given layout inside the store

See clear_breakout_cells for an explanation of breakout cells.

This method has been added in version 0.26.1

(3) Signature: void **set_breakout_cells** (string pattern)

Description: Sets the breakout cell list (as cell name pattern) for the all layouts inside the store

See clear_breakout_cells for an explanation of breakout cells.

This method has been added in version 0.26.1

subcircuit_hierarchy_for_nets=

(1) Signature: void **subcircuit_hierarchy_for_nets=** (bool value)

Description: Sets a value indicating whether to build a subcircuit hierarchy per net

This flag is used to determine the way, net subcircuit hierarchies are built: when true, subcells are created for subcircuits on a net. Otherwise the net shapes are produced flat inside the cell they appear on.

This attribute has been introduced in version 0.28.4

Python specific notes:
The object exposes a writable attribute 'subcircuit_hierarchy_for_nets'. This is the setter.

(2) Signature: void **subcircuit_hierarchy_for_nets=** (bool value)

Description: Gets a value indicating whether to build a subcircuit hierarchy per net

See subcircuit_hierarchy_for_nets= for details.

This attribute has been introduced in version 0.28.4

Python specific notes:
The object exposes a writable attribute 'subcircuit_hierarchy_for_nets'. This is the setter.

text_enlargement

Signature: *[const]* int **text_enlargement**

Description: Gets the text enlargement value.

Python specific notes:
The object exposes a readable attribute 'text_enlargement'. This is the getter.

text_enlargement=

Signature: void **text_enlargement=** (int value)

Description: Sets the text enlargement value

If set to a non-negative value, text objects are converted to boxes with the given enlargement (width = 2 * enlargement). The box centers are identical to the original location of the text. If this value is negative (the default), texts are ignored.

Python specific notes:
The object exposes a writable attribute 'text_enlargement'. This is the setter.

text_property_name

Signature: *[const]* variant **text_property_name**

Description: Gets the text property name.

Python specific notes:
The object exposes a readable attribute 'text_property_name'. This is the getter.

text_property_name=

Signature: void **text_property_name=** (variant name)

Description: Sets the text property name.

If set to a non-null variant, text strings are attached to the generated boxes as properties with this particular name. This option has an effect only if the text_enlargement property is not negative. By default, the name is empty.

Python specific notes:
The object exposes a writable attribute 'text_property_name'. This is the setter.

threads

Signature: *[const]* int **threads**

Description: Gets the number of threads.

Python specific notes:
The object exposes a readable attribute 'threads'. This is the getter.

threads=

Signature: void **threads=** (int n)

Description: Sets the number of threads to allocate for the hierarchical processor

Python specific notes:
The object exposes a writable attribute 'threads'. This is the setter.

wants_all_cells

Signature: *[const]* bool **wants_all_cells**

Description: Gets a flag wether to copy the full hierarchy for the working layouts

This attribute has been introduced in version 0.28.10.

Python specific notes:
The object exposes a readable attribute 'wants_all_cells'. This is the getter.

wants_all_cells=

Signature: void **wants_all_cells=** (bool flag)

Description: Sets a flag wether to copy the full hierarchy for the working layouts

The DeepShapeStore object keeps a copy of the original hierarchy internally for the working layouts. By default, this hierarchy is mapping only non-empty cells. While the operations proceed, more cells may need to be added. This conservative approach saves some memory, but the update operations may reduce overall performance. Setting this flag to 'true' switches to a mode where the full hierarchy is copied always. This will take more memory but may save CPU time.

This attribute has been introduced in version 0.28.10.

Python specific notes:
The object exposes a writable attribute 'wants_all_cells'. This is the setter.

4.170. API reference - Class NetlistCompareLogger

Notation used in Ruby API documentation

Module: db

Description: A base class for netlist comparer event receivers

See GenericNetlistCompareLogger for custom implementations of such receivers.

Public constructors

new NetlistCompareLogger ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

	Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**Signature**: *[static]* new NetlistCompareLogger ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.

4.171. API reference - Class GenericNetlistCompareLogger

Notation used in Ruby API documentation

Module: db

Description: An event receiver for the netlist compare feature.

Class hierarchy: GenericNetlistCompareLogger » NetlistCompareLogger

The NetlistComparer class will send compare events to a logger derived from this class. Use this class to implement your own logger class. You can override on of its methods to receive certain kind of events. This class has been introduced in version 0.26.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[virtual]	void	**begin_circuit**	(const Circuit ptr a, const Circuit ptr b)	This function is called when a new circuit is compared.
[virtual]	void	**begin_netlist**	(const Netlist ptr a, const Netlist ptr b)	This function is called at the beginning of the compare process.
[virtual]	void	**circuit_mismatch**	(const Circuit ptr a, const Circuit ptr b, string msg)	This function is called when circuits can't be compared.
[virtual]	void	**circuit_skipped**	(const Circuit ptr a, const Circuit ptr b, string msg)	This function is called when circuits can't be compared.
[virtual]	void	**device_class_mismat**	(const DeviceClass ptr a, const DeviceClass ptr b, string msg)	This function is called when device classes can't be compared.
[virtual]	void	**device_mismatch**	(const Device ptr a, const Device ptr b, string msg)	This function is called when two devices can't be paired.
[virtual]	void	**end_circuit**	(const Circuit ptr a, const Circuit ptr b, bool matching, string msg)	This function is called at the end of the compare process.
[virtual]	void	**end_netlist**	(const Netlist ptr a, const Netlist ptr b)	This function is called at the end of the compare process.

[virtual]	void	log_entry	(Severity level, string msg)	Issues an entry for the compare log.
[virtual]	void	match_ambiguous_nets	(const Net ptr a, const Net ptr b, string msg)	This function is called when two nets are identified, but this choice is ambiguous.
[virtual]	void	match_devices	(const Device ptr a, const Device ptr b)	This function is called when two devices are identified.
[virtual]	void	match_devices_with_different_device_classes	(const Device ptr a, const Device ptr b)	This function is called when two devices are identified but have different device classes.
[virtual]	void	match_devices_with_	(const Device ptr a, const Device ptr b)	This function is called when two devices are identified but have different parameters.
[virtual]	void	match_nets	(const Net ptr a, const Net ptr b)	This function is called when two nets are identified.
[virtual]	void	match_pins	(const Pin ptr a, const Pin ptr b)	This function is called when two pins are identified.
[virtual]	void	match_subcircuits	(const SubCircuit ptr a, const SubCircuit ptr b)	This function is called when two subcircuits are identified.
[virtual]	void	net_mismatch	(const Net ptr a, const Net ptr b, string msg)	This function is called when a net can't be paired.
[virtual]	void	pin_mismatch	(const Pin ptr a, const Pin ptr b, string msg)	This function is called when two pins can't be paired.
[virtual]	void	subcircuit_mismatch	(const SubCircuit ptr a, const SubCircuit ptr b, string msg)	This function is called when two subcircuits can't be paired.

Public static methods and constants

[static,const]	Severity	Error	Specifies error severity (preferred action is stop)
[static,const]	Severity	Info	Specifies info severity (print if requested, otherwise silent)
[static,const]	Severity	NoSeverity	Specifies no particular severity (default)
[static,const]	Severity	Warning	Specifies warning severity (log with high priority, but do not stop)

Detailed description

Error

Signature: *[static,const]* Severity **Error**

Description: Specifies error severity (preferred action is stop)

Python specific notes:

	The object exposes a readable attribute 'Error'. This is the getter.
Info	**Signature**: *[static,const]* Severity **Info** **Description**: Specifies info severity (print if requested, otherwise silent) **Python specific notes:** The object exposes a readable attribute 'Info'. This is the getter.
NoSeverity	**Signature**: *[static,const]* Severity **NoSeverity** **Description**: Specifies no particular severity (default) **Python specific notes:** The object exposes a readable attribute 'NoSeverity'. This is the getter.
Warning	**Signature**: *[static,const]* Severity **Warning** **Description**: Specifies warning severity (log with high priority, but do not stop) **Python specific notes:** The object exposes a readable attribute 'Warning'. This is the getter.
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

begin_circuit

Signature: *[virtual]* void **begin_circuit** (const Circuit ptr a, const Circuit ptr b)

Description: This function is called when a new circuit is compared.

This compare procedure will run the netlist compare circuit vs. circuit in a bottom-up fashion. Before each circuit is compared, this method is called once with the circuits that are about to be compared. After the circuit has been compared, end_circuit will be called.

In some cases, the compare algorithm will decide that circuits can't be compared. This happens if for some or all subcircuits the pin assignment can't be derived. In this case, circuit_skipped will be called once instead of begin_circuit and end_circuit.

begin_netlist

Signature: *[virtual]* void **begin_netlist** (const Netlist ptr a, const Netlist ptr b)

Description: This function is called at the beginning of the compare process.

This method is called once when the compare run begins.

circuit_mismatch

Signature: *[virtual]* void **circuit_mismatch** (const Circuit ptr a, const Circuit ptr b, string msg)

Description: This function is called when circuits can't be compared.

This method is called when a circuit can't be mapped to a partner in the other netlist. In this case, this method is called with the one circuit and nil for the other circuit.

This method is called instead of begin_circuit and end_circuit.

circuit_skipped

Signature: *[virtual]* void **circuit_skipped** (const Circuit ptr a, const Circuit ptr b, string msg)

Description: This function is called when circuits can't be compared.

If there is a known circuit pair, but the circuits can be compared - for example because subcircuits can't be identified - this method will be called with both circuits.

This method is called instead of begin_circuit and end_circuit.

device_class_mismatch

Signature: *[virtual]* void **device_class_mismatch** (const DeviceClass ptr a, const DeviceClass ptr b, string msg)

Description: This function is called when device classes can't be compared.

This method is called when a device class can't be mapped to a partner in the other netlist. In this case, this method is called with the one device class and nil for the other class.

device_mismatch

Signature: *[virtual]* void **device_mismatch** (const Device ptr a, const Device ptr b, string msg)

Description: This function is called when two devices can't be paired.

This will report the device considered in a or b. The other argument is nil. See match_devices for details.

end_circuit

Signature: *[virtual]* void **end_circuit** (const Circuit ptr a, const Circuit ptr b, bool matching, string msg)

Description: This function is called at the end of the compare process.

The 'matching' argument indicates whether the circuits have been identified as identical. See begin_circuit for details.

end_netlist

Signature: *[virtual]* void **end_netlist** (const Netlist ptr a, const Netlist ptr b)

Description: This function is called at the end of the compare process.

This method is called once when the compare run ended.

log_entry

Signature: *[virtual]* void **log_entry** (Severity level, string msg)

Description: Issues an entry for the compare log.

This method delivers a log message generated during the compare of two circuits. It is called between of begin_circuit and end_circuit.

This method has been added in version 0.28.

match_ambiguous_nets

Signature: *[virtual]* void **match_ambiguous_nets** (const Net ptr a, const Net ptr b, string msg)

Description: This function is called when two nets are identified, but this choice is ambiguous.

This choice is a last-resort fallback to allow continuation of the compare procedure. It is likely that this compare will fail later. Looking for ambiguous nets allows deduction of the origin of this faulty decision. See match_nets for more details.

match_devices

Signature: *[virtual]* void **match_devices** (const Device ptr a, const Device ptr b)

Description: This function is called when two devices are identified.

If two devices are identified as a corresponding pair, this method will be called with both devices. If the devices can be paired, but the device parameters don't match, match_devices_with_different_parameters will be called instead. If the devices can be paired, but the device classes don't match, match_devices_with_different_device_classes will be called instead. If devices can't be matched, device_mismatch will be called with the one device considered and the other device being nil.

match_devices_with_different_device_classes

Signature: *[virtual]* void **match_devices_with_different_device_classes** (const Device ptr a, const Device ptr b)

Description: This function is called when two devices are identified but have different device classes.

See match_devices for details.

match_devices_with_differer

Signature: *[virtual]* void **match_devices_with_different_parameters** (const Device ptr a, const Device ptr b)

Description: This function is called when two devices are identified but have different parameters.

See match_devices for details.

match_nets

Signature: *[virtual]* void **match_nets** (const Net ptr a, const Net ptr b)

Description: This function is called when two nets are identified.

If two nets are identified as a corresponding pair, this method will be called with both nets. If the nets can be paired, but this match is ambiguous, match_ambiguous_nets will be called instead. If nets can't be matched to a partner, net_mismatch will be called.

match_pins	**Signature**: *[virtual]* void **match_pins** (const Pin ptr a, const Pin ptr b) **Description**: This function is called when two pins are identified. If two pins are identified as a corresponding pair, this method will be called with both pins. If pins can't be matched, pin_mismatch will be called with the one pin considered and the other pin being nil.
match_subcircuits	**Signature**: *[virtual]* void **match_subcircuits** (const SubCircuit ptr a, const SubCircuit ptr b) **Description**: This function is called when two subcircuits are identified. If two subcircuits are identified as a corresponding pair, this method will be called with both subcircuits. If subcircuits can't be matched, subcircuit_mismatch will be called with the one subcircuit considered and the other subcircuit being nil.
net_mismatch	**Signature**: *[virtual]* void **net_mismatch** (const Net ptr a, const Net ptr b, string msg) **Description**: This function is called when a net can't be paired. This method will be called, if a net cannot be identified as identical with another net. The corresponding argument will identify the net and source netlist. The other argument will be nil. In some cases, a mismatch is reported with two nets given. This means, nets are known not to match. Still the compare algorithm will proceed as if these nets were equivalent to derive further matches.
pin_mismatch	**Signature**: *[virtual]* void **pin_mismatch** (const Pin ptr a, const Pin ptr b, string msg) **Description**: This function is called when two pins can't be paired. This will report the pin considered in a or b. The other argument is nil. See match_pins for details.
subcircuit_mismatch	**Signature**: *[virtual]* void **subcircuit_mismatch** (const SubCircuit ptr a, const SubCircuit ptr b, string msg) **Description**: This function is called when two subcircuits can't be paired. This will report the subcircuit considered in a or b. The other argument is nil. See match_subcircuits for details.

4.172. API reference - Class NetlistComparer

Notation used in Ruby API documentation

Module: db

Description: Compares two netlists

This class performs a comparison of two netlists. It can be used with an event receiver (logger) to track the errors and net mismatches. Event receivers are derived from class GenericNetlistCompareLogger. The netlist comparer can be configured in different ways. Specific hints can be given for nets, device classes or circuits to improve efficiency and reliability of the graph equivalence deduction algorithm. For example, objects can be marked as equivalent using same_nets, same_circuits etc. The compare algorithm will then use these hints to derive further equivalences. This way, ambiguities can be resolved.

Another configuration option relates to swappable pins of subcircuits. If pins are marked this way, the compare algorithm may swap them to achieve net matching. Swappable pins belong to an 'equivalence group' and can be defined with equivalent_pins.

This class has been introduced in version 0.26.

Public constructors

new NetlistComparer ptr	**new**		Creates a new comparer object.
new NetlistComparer ptr	**new**	(GenericNetlistCompareLogger ptr logger)	Creates a new comparer object.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	bool	**compare**	(const Netlist ptr netlist_a, const Netlist ptr netlist_b)	Compares two netlists.
[const]	bool	**compare**	(const Netlist ptr netlist_a, const Netlist ptr netlist_b, NetlistCompareLogger ptr logger)	Compares two netlists.
[const]	bool	**dont_consider_net_nar**		Gets a value indicating whether net names shall not be considered
	void	**dont_consider_net_names=**	(bool f)	Sets a value indicating whether net names shall not be considered
	void	**equivalent_pins**	(const Circuit ptr circuit_b,	Marks two pins of the given circuit as equivalent (i.e. they can be swapped).

			unsigned long pin_id1, unsigned long pin_id2)	
	void	equivalent_pins	(const Circuit ptr circuit_b, unsigned long[] pin_ids)	Marks several pins of the given circuit as equivalent (i.e. they can be swapped).
	void	join_symmetric_nets	(Circuit ptr circuit)	Joins symmetric nodes in the given circuit.
[const]	unsigned long	max_branch_complexity		Gets the maximum branch complexity
	void	max_branch_complexit	(unsigned long n)	Sets the maximum branch complexity
[const]	unsigned long	max_depth		Gets the maximum search depth
	void	max_depth=	(unsigned long n)	Sets the maximum search depth
	void	max_resistance=	(double threshold)	Excludes all resistor devices with a resistance values higher than the given threshold.
	void	min_capacitance=	(double threshold)	Excludes all capacitor devices with a capacitance values less than the given threshold.
	void	same_circuits	(const Circuit ptr circuit_a, const Circuit ptr circuit_b)	Marks two circuits as identical.
	void	same_device_classes	(const DeviceClass ptr dev_cls_a, const DeviceClass ptr dev_cls_b)	Marks two device classes as identical.
	void	same_nets	(const Net ptr net_a, const Net ptr net_b, bool must_match = false)	Marks two nets as identical.
	void	same_nets	(const Circuit ptr circuit_a, const Circuit ptr circuit_b, const Net ptr net_a, const Net ptr net_b, bool must_match = false)	Marks two nets as identical.
[const]	Circuit ptr[]	unmatched_circuits_a	(Netlist ptr a, Netlist ptr b)	Returns a list of circuits in A for which there is not corresponding circuit in B
[const]	Circuit ptr[]	unmatched_circuits_b	(Netlist ptr a, Netlist ptr b)	Returns a list of circuits in B for which there is not corresponding circuit in A
[const]	bool	with_log		Gets a value indicating that log messages are generated.
	void	with_log=	(bool flag)	Sets a value indicating that log messages are generated.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

compare

(1) Signature: *[const]* bool **compare** (const Netlist ptr netlist_a, const Netlist ptr netlist_b)

Description: Compares two netlists.

This method will perform the actual netlist compare. It will return true if both netlists are identical. If the comparer has been configured with same_nets or similar methods, the objects given there must be located inside 'circuit_a' and 'circuit_b' respectively.

(2) Signature: *[const]* bool **compare** (const Netlist ptr netlist_a, const Netlist ptr netlist_b, NetlistCompareLogger ptr logger)

Description: Compares two netlists.

This method will perform the actual netlist compare using the given logger. It will return true if both netlists are identical. If the comparer has been configured with same_nets or similar methods, the objects given there must be located inside 'circuit_a' and 'circuit_b' respectively.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dont_consider_net_names

Signature: *[const]* bool **dont_consider_net_names**

Description: Gets a value indicating whether net names shall not be considered

See dont_consider_net_names= for details.

Python specific notes:
The object exposes a readable attribute 'dont_consider_net_names'. This is the getter.

dont_consider_net_names=

Signature: void **dont_consider_net_names=** (bool f)

Description: Sets a value indicating whether net names shall not be considered

If this value is set to true, net names will not be considered when resolving ambiguities. Not considering net names usually is more expensive. The default is 'false' indicating that net names will be considered for ambiguity resolution.

This property has been introduced in version 0.26.7.

Python specific notes:
The object exposes a writable attribute 'dont_consider_net_names'. This is the setter.

equivalent_pins

(1) Signature: void **equivalent_pins** (const Circuit ptr circuit_b, unsigned long pin_id1, unsigned long pin_id2)

Description: Marks two pins of the given circuit as equivalent (i.e. they can be swapped).

Only circuits from the second input can be given swappable pins. This will imply the same swappable pins on the equivalent circuit of the first input. To mark multiple pins as swappable, use the version that takes a list of pins.

(2) Signature: void **equivalent_pins** (const Circuit ptr circuit_b, unsigned long[] pin_ids)

Description: Marks several pins of the given circuit as equivalent (i.e. they can be swapped).

Only circuits from the second input can be given swappable pins. This will imply the same swappable pins on the equivalent circuit of the first input. This version is a generic variant of the two-pin version of this method.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

join_symmetric_nets

Signature: void **join_symmetric_nets** (Circuit ptr circuit)

Description: Joins symmetric nodes in the given circuit.

Nodes are symmetrical if swapping them would not modify the circuit. Hence they will carry the same potential and can be connected (joined). This will simplify the circuit and can be applied before device combination to render a schematic-equivalent netlist in some cases (split gate option).

This algorithm will apply the comparer's settings to the symmetry condition (device filtering, device compare tolerances, device class equivalence etc.).

This method has been introduced in version 0.26.4.

max_branch_complexity

Signature: *[const]* unsigned long **max_branch_complexity**

Description: Gets the maximum branch complexity

See max_branch_complexity= for details.

Python specific notes:
The object exposes a readable attribute 'max_branch_complexity'. This is the getter.

max_branch_complexity=

Signature: void **max_branch_complexity=** (unsigned long n)

Description: Sets the maximum branch complexity

This value limits the maximum branch complexity of the backtracking algorithm. The complexity is the accumulated number of branch options with ambiguous net matches. Backtracking will stop when the maximum number of options has been exceeded.

By default, from version 0.27 on the complexity is unlimited and can be reduced in cases where runtimes need to be limited at the cost less elaborate matching evaluation.

As the computational complexity is the square of the branch count, this value should be adjusted carefully.

Python specific notes:
The object exposes a writable attribute 'max_branch_complexity'. This is the setter.

max_depth

Signature: *[const]* unsigned long **max_depth**

Description: Gets the maximum search depth

See max_depth= for details.

Python specific notes:
The object exposes a readable attribute 'max_depth'. This is the getter.

max_depth=

Signature: void **max_depth=** (unsigned long n)

Description: Sets the maximum search depth

This value limits the search depth of the backtracking algorithm to the given number of jumps.

By default, from version 0.27 on the depth is unlimited and can be reduced in cases where runtimes need to be limited at the cost less elaborate matching evaluation.

Python specific notes:
The object exposes a writable attribute 'max_depth'. This is the setter.

max_resistance=

Signature: void **max_resistance=** (double threshold)

Description: Excludes all resistor devices with a resistance values higher than the given threshold.

To reset this constraint, set this attribute to zero.

Python specific notes:
The object exposes a writable attribute 'max_resistance'. This is the setter.

min_capacitance=

Signature: void **min_capacitance=** (double threshold)

Description: Excludes all capacitor devices with a capacitance values less than the given threshold.

To reset this constraint, set this attribute to zero.

Python specific notes:
The object exposes a writable attribute 'min_capacitance'. This is the setter.

new

(1) Signature: *[static]* new NetlistComparer ptr **new**

Description: Creates a new comparer object.

See the class description for more details.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new NetlistComparer ptr **new** (GenericNetlistCompareLogger ptr logger)

Description: Creates a new comparer object.

The logger is a delegate or event receiver which the comparer will send compare events to. See the class description for more details.

Python specific notes:
This method is the default initializer of the object.

same_circuits

Signature: void **same_circuits** (const Circuit ptr circuit_a, const Circuit ptr circuit_b)

Description: Marks two circuits as identical.

	This method makes a circuit circuit_a in netlist a identical to the corresponding circuit circuit_b in netlist b (see compare). By default circuits with the same name are identical.
same_device_classes	**Signature**: void **same_device_classes** (const DeviceClass ptr dev_cls_a, const DeviceClass ptr dev_cls_b) **Description**: Marks two device classes as identical. This makes a device class dev_cls_a in netlist a identical to the corresponding device class dev_cls_b in netlist b (see compare). By default device classes with the same name are identical.
same_nets	**(1) Signature**: void **same_nets** (const Net ptr net_a, const Net ptr net_b, bool must_match = false) **Description**: Marks two nets as identical. This makes a net net_a in netlist a identical to the corresponding net net_b in netlist b (see compare). Otherwise, the algorithm will try to identify nets according to their topology. This method can be used to supply hints to the compare algorithm. It will use these hints to derive further identities. If 'must_match' is true, the nets are required to match. If they don't, an error is reported. The 'must_match' optional argument has been added in version 0.27.3. **(2) Signature**: void **same_nets** (const Circuit ptr circuit_a, const Circuit ptr circuit_b, const Net ptr net_a, const Net ptr net_b, bool must_match = false) **Description**: Marks two nets as identical. This makes a net net_a in netlist a identical to the corresponding net net_b in netlist b (see compare). Otherwise, the algorithm will try to identify nets according to their topology. This method can be used to supply hints to the compare algorithm. It will use these hints to derive further identities. If 'must_match' is true, the nets are required to match. If they don't, an error is reported. This variant allows specifying nil for the nets indicating the nets are mismatched by definition. with 'must_match' this will render a net mismatch error. This variant has been added in version 0.27.3.
unmatched_circuits_a	**Signature**: *[const]* Circuit ptr[] **unmatched_circuits_a** (Netlist ptr a, Netlist ptr b) **Description**: Returns a list of circuits in A for which there is not corresponding circuit in B This list can be used to flatten these circuits so they do not participate in the compare process.
unmatched_circuits_b	**Signature**: *[const]* Circuit ptr[] **unmatched_circuits_b** (Netlist ptr a, Netlist ptr b) **Description**: Returns a list of circuits in B for which there is not corresponding circuit in A This list can be used to flatten these circuits so they do not participate in the compare process.
with_log	**Signature**: *[const]* bool **with_log** **Description**: Gets a value indicating that log messages are generated. See with_log= for details about this flag. This attribute have been introduced in version 0.28. **Python specific notes:** The object exposes a readable attribute 'with_log'. This is the getter.
with_log=	**Signature**: void **with_log=** (bool flag)

Description: Sets a value indicating that log messages are generated.

Log messages may be expensive to compute, hence they can be turned off. By default, log messages are generated.

This attribute have been introduced in version 0.28.

Python specific notes:
The object exposes a writable attribute 'with_log'. This is the setter.

4.173. API reference - Class NetlistCrossReference

Notation used in Ruby API documentation

Module: db

Description: Represents the identity mapping between the objects of two netlists.

Class hierarchy: NetlistCrossReference » NetlistCompareLogger

Sub-classes: NetPairData, DevicePairData, PinPairData, SubCircuitPairData, CircuitPairData, NetTerminalRefPair, NetPinRefPair, NetSubcircuitPinRefPair, Status

The NetlistCrossReference object is a container for the results of a netlist comparison. It implemented the NetlistCompareLogger interface, hence can be used as output for a netlist compare operation (NetlistComparer#compare). It's purpose is to store the results of the compare. It is used in this sense inside the LayoutVsSchematic framework.

The basic idea of the cross reference object is pairing: the netlist comparer will try to identify matching items and store them as pairs inside the cross reference object. If no match is found, a single-sided pair is generated: one item is nil in this case. Beside the items, a status is kept which gives more details about success or failure of the match operation.

Item pairing happens on different levels, reflecting the hierarchy of the netlists. On the top level there are circuits. Inside circuits nets, devices, subcircuits and pins are paired. Nets further contribute their connected items through terminals (for devices), pins (outgoing) and subcircuit pins.

This class has been introduced in version 0.26.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	unsigned long	**circuit_count**		Gets the number of circuit pairs in the cross-reference object.
[iter]	NetlistCrossReference::CircuitPairData	**each_circuit_pair**		Delivers the circuit pairs and their status.
[iter]	NetlistCrossReference::De	**each_device_pair**	(const NetlistCrossReference circuit_pair)	Delivers the device pairs and their status for the given circuit pair.
[iter]	NetlistCrossReference::NetPairData	**each_net_pair**	(const NetlistCrossReference::CircuitPairData circuit_pair)	Delivers the net pairs and their status for the given circuit pair.
[iter]	NetlistCrossReference::Ne	**each_net_pin_pair**	(const NetlistCrossReference net_pair)	Delivers the pin pairs for the given net pair.

[iter]	NetlistCrossReference::NetSubcircuitPinPairOfCircuit	**each_net_subcircuit_pin_pair**	(const NetlistCrossReference::NetPinPair net_pair)	Delivers the subcircuit pin pairs for the given net pair.
[iter]	NetlistCrossReference::Ne	**each_net_terminal_pa**	(const NetlistCrossReference net_pair)	Delivers the device terminal pairs for the given net pair.
[iter]	NetlistCrossReference::PinPairData	**each_pin_pair**	(const NetlistCrossReference::Circuit circuit_pair)	Delivers the pin pairs and their status for the given circuit pair.
[iter]	NetlistCrossReference::Su	**each_subcircuit_pair**	(const NetlistCrossReference circuit_pair)	Delivers the subcircuit pairs and their status for the given circuit pair.
[const]	const Netlist ptr	**netlist_a**		Gets the first netlist which participated in the compare.
[const]	const Netlist ptr	**netlist_b**		Gets the second netlist which participated in the compare.
[const]	const Circuit ptr	**other_circuit_for**	(const Circuit ptr circuit)	Gets the matching other circuit for a given primary circuit.
[const]	const Device ptr	**other_device_for**	(const Device ptr device)	Gets the matching other device for a given primary device.
[const]	const Net ptr	**other_net_for**	(const Net ptr net)	Gets the matching other net for a given primary net.
[const]	const Pin ptr	**other_pin_for**	(const Pin ptr pin)	Gets the matching other pin for a given primary pin.
[const]	const SubCircuit ptr	**other_subcircuit_for**	(const SubCircuit ptr subcircuit)	Gets the matching other subcircuit for a given primary subcircuit.

Public static methods and constants

[static,const]	NetlistCrossReference::Status	**Match**	Enum constant NetlistCrossReference::Match
[static,const]	NetlistCrossReference::Status	**MatchWithWarning**	Enum constant NetlistCrossReference::MatchWithWarning
[static,const]	NetlistCrossReference::Status	**Mismatch**	Enum constant NetlistCrossReference::Mismatch
[static,const]	NetlistCrossReference::Status	**NoMatch**	Enum constant NetlistCrossReference::NoMatch
[static,const]	NetlistCrossReference::Status	**None**	Enum constant NetlistCrossReference::None
[static,const]	NetlistCrossReference::Status	**Skipped**	Enum constant NetlistCrossReference::Skipped

Detailed description

Match

Signature: *[static,const]* NetlistCrossReference::Status **Match**

Description: Enum constant NetlistCrossReference::Match

An exact match exists if this code is present.

Python specific notes:
The object exposes a readable attribute 'Match'. This is the getter.

MatchWithWarning

Signature: *[static,const]* NetlistCrossReference::Status **MatchWithWarning**

Description: Enum constant NetlistCrossReference::MatchWithWarning

If this code is present, a match was found but a warning is issued. For nets, this means that the choice is ambiguous and one, unspecific candidate has been chosen. For devices, this means a device match was established, but parameters or the device class are not matching exactly.

Python specific notes:
The object exposes a readable attribute 'MatchWithWarning'. This is the getter.

Mismatch

Signature: *[static,const]* NetlistCrossReference::Status **Mismatch**

Description: Enum constant NetlistCrossReference::Mismatch

This code means there is a match candidate, but exact identity could not be confirmed.

Python specific notes:
The object exposes a readable attribute 'Mismatch'. This is the getter.

NoMatch

Signature: *[static,const]* NetlistCrossReference::Status **NoMatch**

Description: Enum constant NetlistCrossReference::NoMatch

If this code is present, no match could be found. There is also 'Mismatch' which means there is a candidate, but exact identity could not be confirmed.

Python specific notes:
The object exposes a readable attribute 'NoMatch'. This is the getter.

None

Signature: *[static,const]* NetlistCrossReference::Status **None**

Description: Enum constant NetlistCrossReference::None

No specific status is implied if this code is present.

Python specific notes:
This member is available as 'None_' in Python.
The object exposes a readable attribute 'None_'. This is the getter.

Skipped

Signature: *[static,const]* NetlistCrossReference::Status **Skipped**

Description: Enum constant NetlistCrossReference::Skipped

On circuits this code means that a match has not been attempted because subcircuits of this circuits were not matched. As circuit matching happens bottom-up, all subcircuits must match at least with respect to their pins to allow any parent circuit to be matched.

Python specific notes:
The object exposes a readable attribute 'Skipped'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

circuit_count

Signature: *[const]* unsigned long **circuit_count**

Description: Gets the number of circuit pairs in the cross-reference object.

each_circuit_pair

Signature: *[iter]* NetlistCrossReference::CircuitPairData **each_circuit_pair**

Description: Delivers the circuit pairs and their status.

See the class description for details.

each_device_pair

Signature: *[iter]* NetlistCrossReference::DevicePairData **each_device_pair** (const NetlistCrossReference::CircuitPairData circuit_pair)

Description: Delivers the device pairs and their status for the given circuit pair.

	See the class description for details.
each_net_pair	**Signature**: *[iter]* NetlistCrossReference::NetPairData **each_net_pair** (const NetlistCrossReference::CircuitPairData circuit_pair) **Description**: Delivers the net pairs and their status for the given circuit pair. See the class description for details.
each_net_pin_pair	**Signature**: *[iter]* NetlistCrossReference::NetPinRefPair **each_net_pin_pair** (const NetlistCrossReference::NetPairData net_pair) **Description**: Delivers the pin pairs for the given net pair. For the net pair, lists the pin pairs identified on this net.
each_net_subcircuit_pin_pair	**Signature**: *[iter]* NetlistCrossReference::NetSubcircuitPinRefPair **each_net_subcircuit_pin_pair** (const NetlistCrossReference::NetPairData net_pair) **Description**: Delivers the subcircuit pin pairs for the given net pair. For the net pair, lists the subcircuit pin pairs identified on this net.
each_net_terminal_pair	**Signature**: *[iter]* NetlistCrossReference::NetTerminalRefPair **each_net_terminal_pair** (const NetlistCrossReference::NetPairData net_pair) **Description**: Delivers the device terminal pairs for the given net pair. For the net pair, lists the device terminal pairs identified on this net.
each_pin_pair	**Signature**: *[iter]* NetlistCrossReference::PinPairData **each_pin_pair** (const NetlistCrossReference::CircuitPairData circuit_pair) **Description**: Delivers the pin pairs and their status for the given circuit pair. See the class description for details.
each_subcircuit_pair	**Signature**: *[iter]* NetlistCrossReference::SubCircuitPairData **each_subcircuit_pair** (const NetlistCrossReference::CircuitPairData circuit_pair) **Description**: Delivers the subcircuit pairs and their status for the given circuit pair. See the class description for details.
netlist_a	**Signature**: *[const]* const Netlist ptr **netlist_a** **Description**: Gets the first netlist which participated in the compare. This member may be nil, if the respective netlist is no longer valid. In this case, the netlist cross-reference object cannot be used.
netlist_b	**Signature**: *[const]* const Netlist ptr **netlist_b** **Description**: Gets the second netlist which participated in the compare. This member may be nil, if the respective netlist is no longer valid.In this case, the netlist cross-reference object cannot be used.
other_circuit_for	**Signature**: *[const]* const Circuit ptr **other_circuit_for** (const Circuit ptr circuit) **Description**: Gets the matching other circuit for a given primary circuit. The return value will be nil if no match is found. Otherwise it is the 'b' circuit for circuits from the 'a' netlist and vice versa.

	This method has been introduced in version 0.27.
other_device_for	**Signature**: *[const]* const Device ptr **other_device_for** (const Device ptr device) **Description**: Gets the matching other device for a given primary device. The return value will be nil if no match is found. Otherwise it is the 'b' device for devices from the 'a' netlist and vice versa. This method has been introduced in version 0.27.
other_net_for	**Signature**: *[const]* const Net ptr **other_net_for** (const Net ptr net) **Description**: Gets the matching other net for a given primary net. The return value will be nil if no match is found. Otherwise it is the 'b' net for nets from the 'a' netlist and vice versa.
other_pin_for	**Signature**: *[const]* const Pin ptr **other_pin_for** (const Pin ptr pin) **Description**: Gets the matching other pin for a given primary pin. The return value will be nil if no match is found. Otherwise it is the 'b' pin for pins from the 'a' netlist and vice versa. This method has been introduced in version 0.27.
other_subcircuit_for	**Signature**: *[const]* const SubCircuit ptr **other_subcircuit_for** (const SubCircuit ptr subcircuit) **Description**: Gets the matching other subcircuit for a given primary subcircuit. The return value will be nil if no match is found. Otherwise it is the 'b' subcircuit for subcircuits from the 'a' netlist and vice versa. This method has been introduced in version 0.27.

4.174. API reference - Class NetlistCrossReference::NetPairData

Notation used in Ruby API documentation

Module: db

Description: A net match entry.

This class is equivalent to the class NetlistCrossReference::NetPairData

This object is used to describe the relationship of two nets in a netlist match.

Upon successful match, the first and second members are the matching objects and status is 'Match'. This object is also used to describe non-matches or match errors. In this case, first or second may be nil and status further describes the case.

Public methods

[const]	const Net ptr	**first**	Gets the first object of the relation pair.
[const]	const Net ptr	**second**	Gets the second object of the relation pair.
[const]	NetlistCrossReference::Status	**status**	Gets the status of the relation.

Detailed description

first

Signature: *[const]* const Net ptr **first**

Description: Gets the first object of the relation pair.

The first object is usually the one obtained from the layout-derived netlist. This member can be nil if the pair is describing a non-matching reference object. In this case, the second member is the reference object for which no match was found.

second

Signature: *[const]* const Net ptr **second**

Description: Gets the second object of the relation pair.

The first object is usually the one obtained from the reference netlist. This member can be nil if the pair is describing a non-matching layout object. In this case, the first member is the layout-derived object for which no match was found.

status

Signature: *[const]* NetlistCrossReference::Status **status**

Description: Gets the status of the relation.

This enum described the match status of the relation pair.

4.175. API reference - Class NetlistCrossReference::DevicePairData

Notation used in Ruby API documentation

Module: db

Description: A device match entry.

This class is equivalent to the class NetlistCrossReference::DevicePairData

This object is used to describe the relationship of two devices in a netlist match.

Upon successful match, the first and second members are the matching objects and status is 'Match'. This object is also used to describe non-matches or match errors. In this case, first or second may be nil and status further describes the case.

Public methods

[const]	const Device ptr	**first**	Gets the first object of the relation pair.
[const]	const Device ptr	**second**	Gets the second object of the relation pair.
[const]	NetlistCrossReference::Status	**status**	Gets the status of the relation.

Detailed description

first

Signature: *[const]* const Device ptr **first**

Description: Gets the first object of the relation pair.

The first object is usually the one obtained from the layout-derived netlist. This member can be nil if the pair is describing a non-matching reference object. In this case, the second member is the reference object for which no match was found.

second

Signature: *[const]* const Device ptr **second**

Description: Gets the second object of the relation pair.

The first object is usually the one obtained from the reference netlist. This member can be nil if the pair is describing a non-matching layout object. In this case, the first member is the layout-derived object for which no match was found.

status

Signature: *[const]* NetlistCrossReference::Status **status**

Description: Gets the status of the relation.

This enum described the match status of the relation pair.

4.176. API reference - Class NetlistCrossReference::PinPairData

Notation used in Ruby API documentation

Module: db

Description: A pin match entry.

This class is equivalent to the class NetlistCrossReference::PinPairData

This object is used to describe the relationship of two circuit pins in a netlist match.

Upon successful match, the first and second members are the matching objects and status is 'Match'. This object is also used to describe non-matches or match errors. In this case, first or second may be nil and status further describes the case.

Public methods

[const]	const Pin ptr	**first**	Gets the first object of the relation pair.
[const]	const Pin ptr	**second**	Gets the second object of the relation pair.
[const]	NetlistCrossReference::Status	**status**	Gets the status of the relation.

Detailed description

first

Signature: *[const]* const Pin ptr **first**

Description: Gets the first object of the relation pair.

The first object is usually the one obtained from the layout-derived netlist. This member can be nil if the pair is describing a non-matching reference object. In this case, the second member is the reference object for which no match was found.

second

Signature: *[const]* const Pin ptr **second**

Description: Gets the second object of the relation pair.

The first object is usually the one obtained from the reference netlist. This member can be nil if the pair is describing a non-matching layout object. In this case, the first member is the layout-derived object for which no match was found.

status

Signature: *[const]* NetlistCrossReference::Status **status**

Description: Gets the status of the relation.

This enum described the match status of the relation pair.

4.177. API reference - Class NetlistCrossReference::SubCircuitPairData

Notation used in Ruby API documentation

Module: db

Description: A subcircuit match entry.

This class is equivalent to the class NetlistCrossReference::SubCircuitPairData

This object is used to describe the relationship of two subcircuits in a netlist match.

Upon successful match, the first and second members are the matching objects and status is 'Match'. This object is also used to describe non-matches or match errors. In this case, first or second may be nil and status further describes the case.

Public methods

[const]	const SubCircuit ptr	**first**	Gets the first object of the relation pair.
[const]	const SubCircuit ptr	**second**	Gets the second object of the relation pair.
[const]	NetlistCrossReference::Status	**status**	Gets the status of the relation.

Detailed description

first

Signature: *[const]* const SubCircuit ptr **first**

Description: Gets the first object of the relation pair.

The first object is usually the one obtained from the layout-derived netlist. This member can be nil if the pair is describing a non-matching reference object. In this case, the second member is the reference object for which no match was found.

second

Signature: *[const]* const SubCircuit ptr **second**

Description: Gets the second object of the relation pair.

The first object is usually the one obtained from the reference netlist. This member can be nil if the pair is describing a non-matching layout object. In this case, the first member is the layout-derived object for which no match was found.

status

Signature: *[const]* NetlistCrossReference::Status **status**

Description: Gets the status of the relation.

This enum described the match status of the relation pair.

4.178. API reference - Class NetlistCrossReference::CircuitPairData

Notation used in Ruby API documentation

Module: db

Description: A circuit match entry.

This class is equivalent to the class NetlistCrossReference::CircuitPairData

This object is used to describe the relationship of two circuits in a netlist match.

Upon successful match, the first and second members are the matching objects and status is 'Match'. This object is also used to describe non-matches or match errors. In this case, first or second may be nil and status further describes the case.

Public methods

[const]	const Circuit ptr	**first**	Gets the first object of the relation pair.
[const]	const Circuit ptr	**second**	Gets the second object of the relation pair.
[const]	NetlistCrossReference::Status	**status**	Gets the status of the relation.

Detailed description

first

Signature: *[const]* const Circuit ptr **first**

Description: Gets the first object of the relation pair.

The first object is usually the one obtained from the layout-derived netlist. This member can be nil if the pair is describing a non-matching reference object. In this case, the second member is the reference object for which no match was found.

second

Signature: *[const]* const Circuit ptr **second**

Description: Gets the second object of the relation pair.

The first object is usually the one obtained from the reference netlist. This member can be nil if the pair is describing a non-matching layout object. In this case, the first member is the layout-derived object for which no match was found.

status

Signature: *[const]* NetlistCrossReference::Status **status**

Description: Gets the status of the relation.

This enum described the match status of the relation pair.

4.179. API reference - Class NetlistCrossReference::NetTerminalRefPair

Notation used in Ruby API documentation

Module: db

Description: A match entry for a net terminal pair.

This class is equivalent to the class NetlistCrossReference::NetTerminalRefPair

This object is used to describe the matching terminal pairs or non-matching terminals on a net.

Upon successful match, the first and second members are the matching net objects.Otherwise, either first or second is nil and the other member is the object for which no match was found.

Public methods

[const]	const NetTerminalRef ptr	**first**	Gets the first object of the relation pair.
[const]	const NetTerminalRef ptr	**second**	Gets the second object of the relation pair.

Detailed description

first

Signature: *[const]* const NetTerminalRef ptr **first**

Description: Gets the first object of the relation pair.

The first object is usually the one obtained from the layout-derived netlist. This member can be nil if the pair is describing a non-matching reference object. In this case, the second member is the reference object for which no match was found.

second

Signature: *[const]* const NetTerminalRef ptr **second**

Description: Gets the second object of the relation pair.

The first object is usually the one obtained from the reference netlist. This member can be nil if the pair is describing a non-matching layout object. In this case, the first member is the layout-derived object for which no match was found.

4.180. API reference - Class NetlistCrossReference::NetPinRefPair

Notation used in Ruby API documentation

Module: db

Description: A match entry for a net pin pair.

This class is equivalent to the class NetlistCrossReference::NetPinRefPair

This object is used to describe the matching pin pairs or non-matching pins on a net.

Upon successful match, the first and second members are the matching net objects.Otherwise, either first or second is nil and the other member is the object for which no match was found.

Public methods

[const]	const NetPinRef ptr	**first**	Gets the first object of the relation pair.
[const]	const NetPinRef ptr	**second**	Gets the second object of the relation pair.

Detailed description

first

Signature: *[const]* const NetPinRef ptr **first**

Description: Gets the first object of the relation pair.

The first object is usually the one obtained from the layout-derived netlist. This member can be nil if the pair is describing a non-matching reference object. In this case, the second member is the reference object for which no match was found.

second

Signature: *[const]* const NetPinRef ptr **second**

Description: Gets the second object of the relation pair.

The first object is usually the one obtained from the reference netlist. This member can be nil if the pair is describing a non-matching layout object. In this case, the first member is the layout-derived object for which no match was found.

4.181. API reference - Class NetlistCrossReference::NetSubcircuitPinRefPair

Notation used in Ruby API documentation

Module: db

Description: A match entry for a net subcircuit pin pair.

This class is equivalent to the class NetlistCrossReference::NetSubcircuitPinRefPair

This object is used to describe the matching subcircuit pin pairs or non-matching subcircuit pins on a net.

Upon successful match, the first and second members are the matching net objects.Otherwise, either first or second is nil and the other member is the object for which no match was found.

Public methods

[const]	const NetSubcircuitPinRef ptr	**first**	Gets the first object of the relation pair.
[const]	const NetSubcircuitPinRef ptr	**second**	Gets the second object of the relation pair.

Detailed description

first

Signature: *[const]* const NetSubcircuitPinRef ptr **first**

Description: Gets the first object of the relation pair.

The first object is usually the one obtained from the layout-derived netlist. This member can be nil if the pair is describing a non-matching reference object. In this case, the second member is the reference object for which no match was found.

second

Signature: *[const]* const NetSubcircuitPinRef ptr **second**

Description: Gets the second object of the relation pair.

The first object is usually the one obtained from the reference netlist. This member can be nil if the pair is describing a non-matching layout object. In this case, the first member is the layout-derived object for which no match was found.

4.182. API reference - Class NetlistCrossReference::Status

Notation used in Ruby API documentation

Module: db

Description: This class represents the NetlistCrossReference::Status enum

This class is equivalent to the class NetlistCrossReference::Status

Public constructors

new NetlistCrossReference::Status ptr	**new**	(int i)	Creates an enum from an integer value
new NetlistCrossReference::Status ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const NetlistCrossReference::S other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const NetlistCrossReference::S other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const NetlistCrossReference::S other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	NetlistCrossReference::Status	**Match**	Enum constant NetlistCrossReference::Match
[static,const]	NetlistCrossReference::Status	**MatchWithWarning**	Enum constant NetlistCrossReference::MatchWithWarning
[static,const]	NetlistCrossReference::Status	**Mismatch**	Enum constant NetlistCrossReference::Mismatch

[static,const]	NetlistCrossReference::Status	**NoMatch**	Enum constant NetlistCrossReference::NoMatch
[static,const]	NetlistCrossReference::Status	**None**	Enum constant NetlistCrossReference::None
[static,const]	NetlistCrossReference::Status	**Skipped**	Enum constant NetlistCrossReference::Skipped

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const NetlistCrossReference::Status other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const NetlistCrossReference::Status other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const NetlistCrossReference::Status other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
Match	**Signature**: *[static,const]* NetlistCrossReference::Status **Match** **Description**: Enum constant NetlistCrossReference::Match An exact match exists if this code is present. **Python specific notes:** The object exposes a readable attribute 'Match'. This is the getter.
MatchWithWarning	**Signature**: *[static,const]* NetlistCrossReference::Status **MatchWithWarning** **Description**: Enum constant NetlistCrossReference::MatchWithWarning If this code is present, a match was found but a warning is issued. For nets, this means that the choice is ambiguous and one, unspecific candidate has been chosen. For devices, this means a device match was established, but parameters or the device class are not matching exactly. **Python specific notes:** The object exposes a readable attribute 'MatchWithWarning'. This is the getter.
Mismatch	**Signature**: *[static,const]* NetlistCrossReference::Status **Mismatch** **Description**: Enum constant NetlistCrossReference::Mismatch This code means there is a match candidate, but exact identity could not be confirmed. **Python specific notes:**

The object exposes a readable attribute 'Mismatch'. This is the getter.

NoMatch

Signature: *[static,const]* NetlistCrossReference::Status **NoMatch**

Description: Enum constant NetlistCrossReference::NoMatch

If this code is present, no match could be found. There is also 'Mismatch' which means there is a candidate, but exact identity could not be confirmed.

Python specific notes:
The object exposes a readable attribute 'NoMatch'. This is the getter.

None

Signature: *[static,const]* NetlistCrossReference::Status **None**

Description: Enum constant NetlistCrossReference::None

No specific status is implied if this code is present.

Python specific notes:
This member is available as 'None_' in Python.
The object exposes a readable attribute 'None_'. This is the getter.

Skipped

Signature: *[static,const]* NetlistCrossReference::Status **Skipped**

Description: Enum constant NetlistCrossReference::Skipped

On circuits this code means that a match has not been attempted because subcircuits of this circuits were not matched. As circuit matching happens bottom-up, all subcircuits must match at least with respect to their pins to allow any parent circuit to be matched.

Python specific notes:
The object exposes a readable attribute 'Skipped'. This is the getter.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

new

(1) Signature: *[static]* new NetlistCrossReference::Status ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new NetlistCrossReference::Status ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:

This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.183. API reference - Class LayoutVsSchematic

Notation used in Ruby API documentation

Module: db

Description: A generic framework for doing LVS (layout vs. schematic)

Class hierarchy: LayoutVsSchematic » LayoutToNetlist

This class extends the concept of the netlist extraction from a layout to LVS verification. It does so by adding these concepts to the LayoutToNetlist class:

- A reference netlist. This will be the netlist against which the layout-derived netlist is compared against. See reference and reference=.
- A compare step. During the compare the layout-derived netlist and the reference netlists are compared. The compare results are captured in the cross-reference object. See compare and NetlistComparer for the comparer object.
- A cross-reference. This object (of class NetlistCrossReference) will keep the relations between the objects of the two netlists. It also lists the differences between the netlists. See xref about how to access this object.

The LVS object can be persisted to and from a file in a specific format, so it is sometimes referred to as the "LVS database".

LVS objects can be attached to layout views with LayoutView#add_lvsdb so they become available in the netlist database browser.

This class has been introduced in version 0.26.

Public constructors

new LayoutVsSchematic ptr	**new**	(const RecursiveShapeIterator iter)	Creates a new LVS object with the extractor connected to an original layout
new LayoutVsSchematic ptr	**new**		Creates a new LVS object
new LayoutVsSchematic ptr	**new**	(DeepShapeStore ptr dss)	Creates a new LVS object with the extractor object reusing an existing DeepShapeStore object
new LayoutVsSchematic ptr	**new**	(DeepShapeStore ptr dss, unsigned int layout_index)	Creates a new LVS object with the extractor object reusing an existing DeepShapeStore object
new LayoutVsSchematic ptr	**new**	(string topcell_name, double dbu)	Creates a new LVS object with the extractor object taking a flat DSS

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.

void	**_unmanage**		Marks the object as no longer owned by the script side.
bool	**compare**	(NetlistComparer ptr comparer)	Compare the layout-extracted netlist against the reference netlist using the given netlist comparer.
void	**read**	(string path)	Reads the LVS object from the file.
void	**read_l2n**	(string path)	Reads the LayoutToNetlist part of the object from a file.
Netlist ptr	**reference**		Gets the reference netlist.
void	**reference=**	(Netlist ptr reference_netlist)	Sets the reference netlist.
void	**write**	(string path, bool short_format = false)	Writes the LVS object to a file.
void	**write_l2n**	(string path, bool short_format = false)	Writes the LayoutToNetlist part of the object to a file.
NetlistCrossReference ptr	**xref**		Gets the cross-reference object

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

compare

Signature: bool **compare** (NetlistComparer ptr comparer)

Description: Compare the layout-extracted netlist against the reference netlist using the given netlist comparer.

new

(1) Signature: *[static]* new LayoutVsSchematic ptr **new** (const RecursiveShapeIterator iter)

Description: Creates a new LVS object with the extractor connected to an original layout

This constructor will attach the extractor of the LVS object to an original layout through the shape iterator.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new LayoutVsSchematic ptr **new**

Description: Creates a new LVS object

The main objective for this constructor is to create an object suitable for reading and writing LVS database files.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new LayoutVsSchematic ptr **new** (DeepShapeStore ptr dss)

Description: Creates a new LVS object with the extractor object reusing an existing DeepShapeStore object

See the corresponding constructor of the LayoutToNetlist object for more details.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new LayoutVsSchematic ptr **new** (DeepShapeStore ptr dss, unsigned int layout_index)

Description: Creates a new LVS object with the extractor object reusing an existing DeepShapeStore object

See the corresponding constructor of the LayoutToNetlist object for more details.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new LayoutVsSchematic ptr **new** (string topcell_name, double dbu)

Description: Creates a new LVS object with the extractor object taking a flat DSS

See the corresponding constructor of the LayoutToNetlist object for more details.

Python specific notes:
This method is the default initializer of the object.

read

Signature: void **read** (string path)

Description: Reads the LVS object from the file.

This method employs the native format of KLayout.

read_l2n

Signature: void **read_l2n** (string path)

Description: Reads the LayoutToNetlist part of the object from a file.

This method employs the native format of KLayout.

reference

Signature: Netlist ptr **reference**

Description: Gets the reference netlist.

Python specific notes:
The object exposes a readable attribute 'reference'. This is the getter.

reference=

Signature: void **reference=** (Netlist ptr reference_netlist)

Description: Sets the reference netlist.

This will set the reference netlist used inside compare as the second netlist to compare against the layout-extracted netlist.

The LVS object will take ownership over the netlist - i.e. if it goes out of scope, the reference netlist is deleted.

Python specific notes:
The object exposes a writable attribute 'reference'. This is the setter.

write

Signature: void **write** (string path, bool short_format = false)

Description: Writes the LVS object to a file.

This method employs the native format of KLayout.

write_l2n

Signature: void **write_l2n** (string path, bool short_format = false)

Description: Writes the LayoutToNetlist part of the object to a file.

This method employs the native format of KLayout.

xref

Signature: NetlistCrossReference ptr **xref**

Description: Gets the cross-reference object

The cross-reference object is created while comparing the layout-extracted netlist against the reference netlist - i.e. during compare. Before compare is called, this object is nil. It holds the results of the comparison - a cross-reference between the nets and other objects in the match case and a listing of non-matching nets and other objects for the non-matching cases. See NetlistCrossReference for more details.

4.184. API reference - Class TextFilter

Notation used in Ruby API documentation

Module: db

Description: A generic text filter adaptor

Text filters are an efficient way to filter texts from a Texts collection. To apply a filter, derive your own filter class and pass an instance to Texts#filter or Texts#filtered method.

Conceptually, these methods take each text from the collection and present it to the filter's 'selected' method. Based on the result of this evaluation, the text is kept or discarded.

The magic happens when deep mode text collections are involved. In that case, the filter will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the filter behaves. You need to configure the filter by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using the filter.

You can skip this step, but the filter algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

Here is some example that filters texts with a given string length:

```
class TextStringLengthFilter < RBA::TextFilter

  # Constructor
  def initialize(string_length)
    self.is_isotropic_and_scale_invariant   # orientation and scale do not matter
    @string_length = string_length
  end

  # Select texts with given string length
  def selected(text)
    return text.string.size == @string_length
  end

end

texts = ... # some Texts object
with_length_3 = edges.filtered(TextStringLengthFilter::new(3))
```

This class has been introduced in version 0.29.

Public constructors

new TextFilter ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.

	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**is_isotropic**		Indicates that the filter has isotropic properties
	void	**is_isotropic_and_scale_invariant**		Indicates that the filter is isotropic and scale invariant
	void	**is_scale_invariant**		Indicates that the filter is scale invariant
[virtual,const]	bool	**selected**	(const Text text)	Selects a text
	void	**wants_variants=**	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	**wants_variants?**		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) filters are area or perimeter filters. The area or perimeter of a polygon depends on the scale, but not on the orientation of the polygon.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) filter is the square selector. Whether a polygon is a square or not does not depend on the polygon's orientation nor scale.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) filter is the bounding box aspect ratio (height/width) filter. The definition of heigh and width depends on the orientation, but the ratio is independent on scale.

new

Signature: *[static]* new TextFilter ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

selected

Signature: *[virtual,const]* bool **selected** (const Text text)

Description: Selects a text

This method is the actual payload. It needs to be reimplemented in a derived class. It needs to analyze the text and return 'true' if it should be kept and 'false' if it should be discarded.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.185. API reference - Class TextOperator

Notation used in Ruby API documentation

Module: db

Description: A generic text operator

Text processors are an efficient way to process texts from an text collection. To apply a processor, derive your own operator class and pass an instance to the Texts#processed or Texts#process method.

Conceptually, these methods take each text from the edge pair collection and present it to the operator's 'process' method. The result of this call is a list of zero to many output texts derived from the input text. The output text collection is the sum over all these individual results.

The magic happens when deep mode text collections are involved. In that case, the processor will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the operator behaves. You need to configure the operator by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using it.

You can skip this step, but the processor algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

Here is some example that replaces the text string:

```
class ReplaceTextString < RBA::TextOperator

  # Constructor
  def initialize
    self.is_isotropic_and_scale_invariant   # orientation and scale do not matter
  end

  # Replaces the string by a number representing the string length
  def process(text)
    new_text = text.dup   # need a copy as we cannot modify the text passed
    new_text.string = text.string.size.to_s
    return [ new_text ]
  end

end

texts = ... # some Texts object
modified = texts.processed(ReplaceTextString::new)
```

This class has been introduced in version 0.29.

Public constructors

new TextOperator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference

	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
	void	is_isotropic		Indicates that the filter has isotropic properties
	void	is_isotropic_and_scale_invariant		Indicates that the filter is isotropic and scale invariant
	void	is_scale_invariant		Indicates that the filter is scale invariant
[virtual,const]	Text[]	process	(const Text shape)	Processes a shape
	void	wants_variants=	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	wants_variants?		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) processors are size or shrink operators. Size or shrink is not dependent on orientation unless size or shrink needs to be different in x and y direction.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) processor is the convex decomposition operator. The decomposition of a polygon into convex parts is an operation that is not depending on scale nor orientation.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) processor is the rotation operator. Rotation is not depending on scale, but on the original orientation as mirrored versions need to be rotated differently.

new

Signature: *[static]* new TextOperator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

process

Signature: *[virtual,const]* Text[] **process** (const Text shape)

Description: Processes a shape

This method is the actual payload. It needs to be reimplemented in a derived class. If needs to process the input shape and deliver a list of output shapes. The output list may be empty to entirely discard the input shape. It may also contain more than a single shape. In that case, the number of total shapes may grow during application of the processor.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.186. API reference - Class TextToPolygonOperator

Notation used in Ruby API documentation

Module: db

Description: A generic text-to-polygon operator

Text processors are an efficient way to process texts from an text collection. To apply a processor, derive your own operator class and pass an instance to the Texts#processed method.

Conceptually, these methods take each text from the text collection and present it to the operator's 'process' method. The result of this call is a list of zero to many output polygons derived from the input text. The output region is the sum over all these individual results.

The magic happens when deep mode text collections are involved. In that case, the processor will use as few calls as possible and exploit the hierarchical compression if possible. It needs to know however, how the operator behaves. You need to configure the operator by calling is_isotropic, is_scale_invariant or is_isotropic_and_scale_invariant before using it.

You can skip this step, but the processor algorithm will assume the worst case then. This usually leads to cell variant formation which is not always desired and blows up the hierarchy.

For a basic example see the TextOperator class, with the exception that this incarnation delivers polygons.

This class has been introduced in version 0.29.

Public constructors

new TextToPolygonOperator ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**is_isotropic**		Indicates that the filter has isotropic properties
	void	**is_isotropic_and_scale_invariant**		Indicates that the filter is isotropic and scale invariant
	void	**is_scale_invariant**		Indicates that the filter is scale invariant
[virtual,const]	Polygon[]	**process**	(const Text shape)	Processes a shape
	void	**wants_variants=**	(bool flag)	Sets a value indicating whether the filter prefers cell variants
[const]	bool	**wants_variants?**		Gets a value indicating whether the filter prefers cell variants

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_isotropic

Signature: void **is_isotropic**

Description: Indicates that the filter has isotropic properties

Call this method before using the filter to indicate that the selection is independent of the orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

Examples for isotropic (polygon) processors are size or shrink operators. Size or shrink is not dependent on orientation unless size or shrink needs to be different in x and y direction.

is_isotropic_and_scale_invariant

Signature: void **is_isotropic_and_scale_invariant**

Description: Indicates that the filter is isotropic and scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale and orientation of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for such a (polygon) processor is the convex decomposition operator. The decomposition of a polygon into convex parts is an operation that is not depending on scale nor orientation.

is_scale_invariant

Signature: void **is_scale_invariant**

Description: Indicates that the filter is scale invariant

Call this method before using the filter to indicate that the selection is independent of the scale of the shape. This helps the filter algorithm optimizing the filter run, specifically in hierarchical mode.

An example for a scale invariant (polygon) processor is the rotation operator. Rotation is not depending on scale, but on the original orientation as mirrored versions need to be rotated differently.

new

Signature: *[static]* new TextToPolygonOperator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

process

Signature: *[virtual,const]* Polygon[] **process** (const Text shape)

Description: Processes a shape

This method is the actual payload. It needs to be reimplemented in a derived class. If needs to process the input shape and deliver a list of output shapes. The output list may be empty to entirely discard the input shape. It may also contain more than a single shape. In that case, the number of total shapes may grow during application of the processor.

wants_variants=

Signature: void **wants_variants=** (bool flag)

Description: Sets a value indicating whether the filter prefers cell variants

This flag must be set before using this filter for hierarchical applications (deep mode). It tells the filter implementation whether cell variants should be created (true, the default) or shape propagation will be applied (false).

This decision needs to be made, if the filter indicates that it will deliver different results for scaled or rotated versions of the shape (see is_isotropic and the other hints). If a cell is present with different qualities - as seen from the top cell - the respective instances need to be differentiated. Cell variant formation is one way, shape propagation the other way. Typically, cell variant formation is less expensive, but the hierarchy will be modified.

Python specific notes:
The object exposes a writable attribute 'wants_variants'. This is the setter.

wants_variants?

Signature: *[const]* bool **wants_variants?**

Description: Gets a value indicating whether the filter prefers cell variants

See wants_variants= for details.

Python specific notes:
The object exposes a readable attribute 'wants_variants'. This is the getter.

4.187. API reference - Class Texts

Notation used in Ruby API documentation

Module: db

Description: Texts (a collection of texts)

Class hierarchy: Texts » ShapeCollection

Text objects are useful as labels for net names, to identify certain regions and to specify specific locations in general. Text collections provide a way to store - also in a hierarchical fashion - and manipulate a collection of text objects.

Text objects can be turned into polygons by creating small boxes around the texts (polygons). Texts can also be turned into dot-like edges (edges). Texts can be filtered by string, either by matching against a fixed string (with_text) or a glob-style pattern (with_match).

Text collections can be filtered geometrically against a polygon Region using interacting or non-interacting. Vice versa, texts can be used to select polygons from a Region using pull_interacting.

Beside that, text collections can be transformed, flattened and combined, similar to EdgePairs.

This class has been introduced in version 0.27.

Public constructors

new Texts ptr	**new**		Default constructor
new Texts ptr	**new**	(Text[] array)	Constructor from an text array
new Texts ptr	**new**	(const Text text)	Constructor from a single edge pair object
new Texts ptr	**new**	(const Shapes shapes)	Shapes constructor
new Texts ptr	**new**	(const RecursiveShapeIterator shape_iterator)	Constructor from a hierarchical shape set
new Texts ptr	**new**	(const RecursiveShapeIterator shape_iterator, const ICplxTrans trans)	Constructor from a hierarchical shape set with a transformation
new Texts ptr	**new**	(const RecursiveShapeIterator shape_iterator, DeepShapeStore dss)	Creates a hierarchical text collection from an original layer
new Texts ptr	**new**	(const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, const ICplxTrans trans)	Creates a hierarchical text collection from an original layer with a transformation

Public methods

[const]	Texts	**&**	(const Region other)	Returns the texts from this text collection which are inside or on the edge of polygons from the given region
[const]	Texts	**+**	(const Texts other)	Returns the combined text collection of self and the other one
	Texts	**+=**	(const Texts other)	Adds the texts of the other text collection to self

[const]	Texts	**-**	(const Region other)	Returns the texts from this text collection which are not inside or on the edge of polygons from the given region
[const]	const Text ptr	**[]**	(unsigned long n)	Returns the nth text
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Texts other)	Assigns another object to self
[const]	Box	**bbox**		Return the bounding box of the text collection
	void	**clear**		Clears the text collection
[const]	unsigned long	**count**		Returns the (flat) number of texts in the text collection
[const]	unsigned long	**data_id**		Returns the data ID (a unique identifier for the underlying data storage)
	void	**disable_progress**		Disable progress reporting
[const]	new Texts ptr	**dup**		Creates a copy of self
[const,iter]	Text	**each**		Returns each text of the text collection
[const]	Edges	**edges**		Returns dot-like edges for the texts
	void	**enable_progress**	(string label)	Enable progress reporting
	void	**enable_properties**		Enables properties for the given container.
[const]	Region	**extents**	(int d = 1)	Returns a region with the enlarged bounding boxes of the texts
[const]	Region	**extents**	(int dx, int dy)	Returns a region with the enlarged bounding boxes of the texts
	void	**filter**	(const TextFilter ptr filter)	Applies a generic filter in place (replacing the texts from the Texts collection)
	void	**filter_properties**	(variant[] keys)	Filters properties by certain keys.

[const]	Texts	**filtered**	(const TextFilter ptr filtered)	Applies a generic filter and returns a filtered copy
	void	**flatten**		Explicitly flattens an text collection
[const]	bool	**has_valid_texts?**		Returns true if the text collection is flat and individual texts can be accessed randomly
[const]	unsigned long	**hier_count**		Returns the (hierarchical) number of texts in the text collection
	void	**insert**	(const Text text)	Inserts a text into the collection
	void	**insert**	(const Texts texts)	Inserts all texts from the other text collection into this collection
[const]	void	**insert_into**	(Layout ptr layout, unsigned int cell_index, unsigned int layer)	Inserts this texts into the given layout, below the given cell and into the given layer.
[const]	void	**insert_into_as_polygons**	(Layout ptr layout, unsigned int cell_index, unsigned int layer, int e)	Inserts this texts into the given layout, below the given cell and into the given layer.
[const]	Texts	**interacting**	(const Region other)	Returns the texts from this text collection which are inside or on the edge of polygons from the given region
[const]	bool	**is_deep?**		Returns true if the edge pair collection is a deep (hierarchical) one
[const]	bool	**is_empty?**		Returns true if the collection is empty
[const]	Texts	**join**	(const Texts other)	Returns the combined text collection of self and the other one
	Texts	**join_with**	(const Texts other)	Adds the texts of the other text collection to self
	void	**map_properties**	(map<variant,variant> key_map)	Maps properties by name key.
	Texts	**move**	(const Vector p)	Moves the text collection
	Texts	**move**	(int x, int y)	Moves the text collection
[const]	Texts	**moved**	(const Vector p)	Returns the moved text collection (does not modify self)
[const]	Texts	**moved**	(int x, int y)	Returns the moved edge pair collection (does not modify self)
[const]	Texts	**not_interacting**	(const Region other)	Returns the texts from this text collection which are not inside or on the edge of polygons from the given region

[const]	Region	**polygons**	(int e = 1)	Converts the edge pairs to polygons
	void	**process**	(const TextOperator ptr process)	Applies a generic text processor in place (replacing the texts from the text collection)
[const]	Texts	**processed**	(const TextOperator ptr processed)	Applies a generic text processor and returns a processed copy
[const]	Region	**processed**	(const TextToPolygonOperato ptr processed)	Applies a generic text-to-polygon processor and returns a region with the results
[const]	Region	**pull_interacting**	(const Region other)	Returns all polygons of "other" which are including texts of this text set
	void	**remove_properties**		Removes properties for the given container.
	Texts	**select_interacting**	(const Region other)	Selects the texts from this text collection which are inside or on the edge of polygons from the given region
	Texts	**select_not_interactir**	(const Region other)	Selects the texts from this text collection which are not inside or on the edge of polygons from the given region
	void	**swap**	(Texts other)	Swap the contents of this collection with the contents of another collection
[const]	string	**to_s**		Converts the text collection to a string
[const]	string	**to_s**	(unsigned long max_count)	Converts the text collection to a string
	Texts	**transform**	(const Trans t)	Transform the text collection (modifies self)
	Texts	**transform**	(const ICplxTrans t)	Transform the text collection with a complex transformation (modifies self)
[const]	Texts	**transformed**	(const Trans t)	Transform the edge pair collection
[const]	Texts	**transformed**	(const ICplxTrans t)	Transform the text collection with a complex transformation
[const]	Texts	**with_match**	(string pattern, bool inverse)	Filter the text by glob pattern
[const]	Texts	**with_text**	(string text, bool inverse)	Filter the text by text string
[const]	void	**write**	(string filename)	Writes the region to a file

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead

[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	unsigned long	**size**		Use of this method is deprecated. Use count instead
	Texts	**transform_icplx**	(const ICplxTrans t)	Use of this method is deprecated. Use transform instead
[const]	Texts	**transformed_icplx**	(const ICplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

&

Signature: *[const]* Texts **&** (const Region other)

Description: Returns the texts from this text collection which are inside or on the edge of polygons from the given region

Returns: A new text collection containing the texts inside or on the edge of polygons from the region

+

Signature: *[const]* Texts **+** (const Texts other)

Description: Returns the combined text collection of self and the other one

Returns: The resulting text collection

This operator adds the texts of the other collection to self and returns a new combined set.

The 'join' alias has been introduced in version 0.28.12.

+=

Signature: Texts **+=** (const Texts other)

Description: Adds the texts of the other text collection to self

Returns: The text collection after modification (self)

This operator adds the texts of the other collection to self.

Note that in Ruby, the '+=' operator actually does not exist, but is emulated by '+' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'join_with' instead.

The 'join_with' alias has been introduced in version 0.28.12.

-

Signature: *[const]* Texts **-** (const Region other)

Description: Returns the texts from this text collection which are not inside or on the edge of polygons from the given region

Returns: A new text collection containing the texts not inside or on the edge of polygons from the region

[]

Signature: *[const]* const Text ptr **[]** (unsigned long n)

Description: Returns the nth text

This method returns nil if the index is out of range. It is available for flat texts only - i.e. those for which has_valid_texts? is true. Use flatten to explicitly flatten an text collection.

The each iterator is the more general approach to access the texts.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Texts other)

Description: Assigns another object to self

bbox

Signature: *[const]* Box **bbox**

Description: Return the bounding box of the text collection

The bounding box is the box enclosing all origins of all texts.

clear

Signature: void **clear**

Description: Clears the text collection

count

Signature: *[const]* unsigned long **count**

Description: Returns the (flat) number of texts in the text collection

The count is computed 'as if flat', i.e. texts inside a cell are multiplied by the number of times a cell is instantiated.

Starting with version 0.27, the method is called 'count' for consistency with Region. 'size' is still provided as an alias.

Python specific notes:
This method is also available as 'len(object)'.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

data_id

Signature: *[const]* unsigned long **data_id**

Description: Returns the data ID (a unique identifier for the underlying data storage)

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

disable_progress

Signature: void **disable_progress**

Description: Disable progress reporting

Calling this method will disable progress reporting. See enable_progress.

dup

Signature: *[const]* new Texts ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each

Signature: *[const,iter]* Text **each**

Description: Returns each text of the text collection

Python specific notes:
This method enables iteration of the object.

edges

Signature: *[const]* Edges **edges**

Description: Returns dot-like edges for the texts

Returns: An edge collection containing the individual, dot-like edges

enable_progress

Signature: void **enable_progress** (string label)

Description: Enable progress reporting

After calling this method, the text collection will report the progress through a progress bar while expensive operations are running. The label is a text which is put in front of the progress bar. Using a progress bar will imply a performance penalty of a few percent typically.

enable_properties

Signature: void **enable_properties**

Description: Enables properties for the given container.

This method has an effect mainly on original layers and will import properties from such layers. By default, properties are not enabled on original layers. Alternatively you can apply filter_properties or map_properties to enable properties with a specific name key.

This method has been introduced in version 0.28.4.

extents

(1) Signature: *[const]* Region **extents** (int d = 1)

Description: Returns a region with the enlarged bounding boxes of the texts

Text bounding boxes are point-like boxes which vanish unless an enlargement of >0 is specified. The bounding box is centered at the text's location. The boxes will not be merged, so it is possible to determine overlaps of these boxes for example.

(2) Signature: *[const]* Region **extents** (int dx, int dy)

Description: Returns a region with the enlarged bounding boxes of the texts

This method acts like the other version of extents, but allows giving different enlargements for x and y direction.

filter

Signature: void **filter** (const TextFilter ptr filter)

Description: Applies a generic filter in place (replacing the texts from the Texts collection)

See TextFilter for a description of this feature.

This method has been introduced in version 0.29.

filter_properties

Signature: void **filter_properties** (variant[] keys)

Description: Filters properties by certain keys.

Calling this method on a container will reduce the properties to values with name keys from the 'keys' list. As a side effect, this method enables properties on original layers.

This method has been introduced in version 0.28.4.

filtered

Signature: *[const]* Texts **filtered** (const TextFilter ptr filtered)

Description: Applies a generic filter and returns a filtered copy

See TextFilter for a description of this feature.

This method has been introduced in version 0.29.

flatten

Signature: void **flatten**

Description: Explicitly flattens an text collection

If the collection is already flat (i.e. has_valid_texts? returns true), this method will not change the collection.

has_valid_texts?

Signature: *[const]* bool **has_valid_texts?**

Description: Returns true if the text collection is flat and individual texts can be accessed randomly

hier_count

Signature: *[const]* unsigned long **hier_count**

Description: Returns the (hierarchical) number of texts in the text collection

The count is computed 'hierarchical', i.e. texts inside a cell are counted once even if the cell is instantiated multiple times.

This method has been introduced in version 0.27.

insert

(1) Signature: void **insert** (const Text text)

Description: Inserts a text into the collection

(2) Signature: void **insert** (const Texts texts)

Description: Inserts all texts from the other text collection into this collection

insert_into

Signature: *[const]* void **insert_into** (Layout ptr layout, unsigned int cell_index, unsigned int layer)

Description: Inserts this texts into the given layout, below the given cell and into the given layer.

If the text collection is a hierarchical one, a suitable hierarchy will be built below the top cell or and existing hierarchy will be reused.

insert_into_as_polygons

Signature: *[const]* void **insert_into_as_polygons** (Layout ptr layout, unsigned int cell_index, unsigned int layer, int e)

Description: Inserts this texts into the given layout, below the given cell and into the given layer.

If the text collection is a hierarchical one, a suitable hierarchy will be built below the top cell or and existing hierarchy will be reused.

The texts will be converted to polygons with the enlargement value given be 'e'. See polygon or extents for details.

interacting

Signature: *[const]* Texts **interacting** (const Region other)

Description: Returns the texts from this text collection which are inside or on the edge of polygons from the given region

Returns: A new text collection containing the texts inside or on the edge of polygons from the region

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_deep?

Signature: *[const]* bool **is_deep?**

Description: Returns true if the edge pair collection is a deep (hierarchical) one

is_empty?

Signature: *[const]* bool **is_empty?**

Description: Returns true if the collection is empty

join

Signature: *[const]* Texts **join** (const Texts other)

Description: Returns the combined text collection of self and the other one

Returns: The resulting text collection

This operator adds the texts of the other collection to self and returns a new combined set.

The 'join' alias has been introduced in version 0.28.12.

join_with

Signature: Texts **join_with** (const Texts other)

Description: Adds the texts of the other text collection to self

Returns: The text collection after modification (self)

This operator adds the texts of the other collection to self.

Note that in Ruby, the '+=' operator actually does not exist, but is emulated by '+' followed by an assignment. This is less efficient than the in-place operation, so it is recommended to use 'join_with' instead.

The 'join_with' alias has been introduced in version 0.28.12.

map_properties

Signature: void **map_properties** (map<variant,variant> key_map)

Description: Maps properties by name key.

Calling this method on a container will reduce the properties to values with name keys from the 'keys' hash and renames the properties. Properties not listed in the key map will be removed. As a side effect, this method enables properties on original layers.

This method has been introduced in version 0.28.4.

move

(1) Signature: Texts **move** (const Vector p)

Description: Moves the text collection

p: The distance to move the texts.

Returns: The moved texts (self).

Moves the texts by the given offset and returns the moved text collection. The text collection is overwritten.

(2) Signature: Texts **move** (int x, int y)

Description: Moves the text collection

x: The x distance to move the texts.

y: The y distance to move the texts.

Returns: The moved texts (self).

Moves the edge pairs by the given offset and returns the moved texts. The edge pair collection is overwritten.

moved

(1) Signature: *[const]* Texts **moved** (const Vector p)

Description: Returns the moved text collection (does not modify self)

p: The distance to move the texts.

Returns: The moved texts.

Moves the texts by the given offset and returns the moved texts. The text collection is not modified.

(2) Signature: *[const]* Texts **moved** (int x, int y)

Description: Returns the moved edge pair collection (does not modify self)

x: The x distance to move the texts.

y: The y distance to move the texts.

Returns: The moved texts.

Moves the texts by the given offset and returns the moved texts. The text collection is not modified.

new

(1) Signature: *[static]* new Texts ptr **new**

Description: Default constructor

This constructor creates an empty text collection.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Texts ptr **new** (Text[] array)

Description: Constructor from an text array

This constructor creates an text collection from an array of Text objects.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Texts ptr **new** (const Text text)

Description: Constructor from a single edge pair object

This constructor creates an text collection with a single text.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Texts ptr **new** (const Shapes shapes)

Description: Shapes constructor

This constructor creates an text collection from a Shapes collection.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new Texts ptr **new** (const RecursiveShapeIterator shape_iterator)

Description: Constructor from a hierarchical shape set

This constructor creates a text collection from the shapes delivered by the given recursive shape iterator. Only texts are taken from the shape set and other shapes are ignored. This method allows feeding the text collection from a hierarchy of cells.

```
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
r = RBA::Texts::new(layout.begin_shapes(cell, layer))
```

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new Texts ptr **new** (const RecursiveShapeIterator shape_iterator, const ICplxTrans trans)

Description: Constructor from a hierarchical shape set with a transformation

This constructor creates a text collection from the shapes delivered by the given recursive shape iterator. Only texts are taken from the shape set and other shapes are ignored. The given transformation is applied to each text taken. This method allows feeding the text collection from a hierarchy of cells. The transformation is useful to scale to a specific database unit for example.

```
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
dbu    = 0.1 # the target database unit
r = RBA::Texts::new(layout.begin_shapes(cell, layer),
 RBA::ICplxTrans::new(layout.dbu / dbu))
```

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new Texts ptr **new** (const RecursiveShapeIterator shape_iterator, DeepShapeStore dss)

Description: Creates a hierarchical text collection from an original layer

This constructor creates a text collection from the shapes delivered by the given recursive shape iterator. This version will create a hierarchical text collection which supports hierarchical operations.

```
dss    = RBA::DeepShapeStore::new
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
r = RBA::Texts::new(layout.begin_shapes(cell, layer))
```

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new Texts ptr **new** (const RecursiveShapeIterator shape_iterator, DeepShapeStore dss, const ICplxTrans trans)

Description: Creates a hierarchical text collection from an original layer with a transformation

This constructor creates a text collection from the shapes delivered by the given recursive shape iterator. This version will create a hierarchical text collection which supports hierarchical operations. The transformation is useful to scale to a specific database unit for example.

```
dss    = RBA::DeepShapeStore::new
layout = ... # a layout
cell   = ... # the index of the initial cell
layer  = ... # the index of the layer from where to take the shapes from
dbu    = 0.1 # the target database unit
r = RBA::Texts::new(layout.begin_shapes(cell, layer),
 RBA::ICplxTrans::new(layout.dbu / dbu))
```

Python specific notes:
This method is the default initializer of the object.

not_interacting

Signature: *[const]* Texts **not_interacting** (const Region other)

Description: Returns the texts from this text collection which are not inside or on the edge of polygons from the given region

Returns: A new text collection containing the texts not inside or on the edge of polygons from the region

polygons

Signature: *[const]* Region **polygons** (int e = 1)

Description: Converts the edge pairs to polygons

This method creates polygons from the texts. This is equivalent to calling extents.

process

Signature: void **process** (const TextOperator ptr process)

Description: Applies a generic text processor in place (replacing the texts from the text collection)

See TextProcessor for a description of this feature.

This method has been introduced in version 0.29.

processed

(1) Signature: *[const]* Texts **processed** (const TextOperator ptr processed)

Description: Applies a generic text processor and returns a processed copy

See TextProcessor for a description of this feature.

This method has been introduced in version 0.29.

(2) Signature: *[const]* Region **processed** (const TextToPolygonOperator ptr processed)

Description: Applies a generic text-to-polygon processor and returns a region with the results

See TextToPolygonProcessor for a description of this feature.

This method has been introduced in version 0.29.

pull_interacting

Signature: *[const]* Region **pull_interacting** (const Region other)

Description: Returns all polygons of "other" which are including texts of this text set

Returns: The region after the polygons have been selected (from other)

The "pull_..." method is similar to "select_..." but works the opposite way: it selects shapes from the argument region rather than self. In a deep (hierarchical) context the output region will be hierarchically aligned with self, so the "pull_..." method provide a way for re-hierarchization.

Merged semantics applies for the polygon region.

remove_properties

Signature: void **remove_properties**

Description: Removes properties for the given container.

This will remove all properties on the given container.

This method has been introduced in version 0.28.4.

select_interacting

Signature: Texts **select_interacting** (const Region other)

Description: Selects the texts from this text collection which are inside or on the edge of polygons from the given region

Returns: A text collection after the texts have been selected (self)

In contrast to interacting, this method will modify self.

select_not_interacting

Signature: Texts **select_not_interacting** (const Region other)

Description: Selects the texts from this text collection which are not inside or on the edge of polygons from the given region

Returns: A text collection after the texts have been selected (self)

In contrast to interacting, this method will modify self.

size

Signature: *[const]* unsigned long **size**

Description: Returns the (flat) number of texts in the text collection

Use of this method is deprecated. Use count instead

The count is computed 'as if flat', i.e. texts inside a cell are multiplied by the number of times a cell is instantiated.

Starting with version 0.27, the method is called 'count' for consistency with Region. 'size' is still provided as an alias.

Python specific notes:
This method is also available as 'len(object)'.

swap

Signature: void **swap** (Texts other)

Description: Swap the contents of this collection with the contents of another collection

This method is useful to avoid excessive memory allocation in some cases. For managed memory languages such as Ruby, those cases will be rare.

to_s

(1) Signature: *[const]* string **to_s**

Description: Converts the text collection to a string

The length of the output is limited to 20 texts to avoid giant strings on large collections. For full output use "to_s" with a maximum count parameter.

Python specific notes:
This method is also available as 'str(object)'.

(2) Signature: *[const]* string **to_s** (unsigned long max_count)

Description: Converts the text collection to a string

This version allows specification of the maximum number of texts contained in the string.

transform

(1) Signature: Texts **transform** (const Trans t)

Description: Transform the text collection (modifies self)

t: The transformation to apply.

Returns: The transformed text collection.

Transforms the text collection with the given transformation. This version modifies the text collection and returns a reference to self.

(2) Signature: Texts **transform** (const ICplxTrans t)

Description: Transform the text collection with a complex transformation (modifies self)

t: The transformation to apply.

Returns: The transformed text collection.

Transforms the text collection with the given transformation. This version modifies the text collection and returns a reference to self.

transform_icplx

Signature: Texts **transform_icplx** (const ICplxTrans t)

Description: Transform the text collection with a complex transformation (modifies self)

t: The transformation to apply.

Returns: The transformed text collection.

Use of this method is deprecated. Use transform instead

Transforms the text collection with the given transformation. This version modifies the text collection and returns a reference to self.

transformed

(1) Signature: *[const]* Texts **transformed** (const Trans t)

Description: Transform the edge pair collection

t: The transformation to apply.

Returns: The transformed texts.

Transforms the texts with the given transformation. Does not modify the edge pair collection but returns the transformed texts.

(2) Signature: *[const]* Texts **transformed** (const ICplxTrans t)

Description: Transform the text collection with a complex transformation

t: The transformation to apply.

Returns: The transformed texts.

Transforms the text with the given complex transformation. Does not modify the text collection but returns the transformed texts.

transformed_icplx

Signature: *[const]* Texts **transformed_icplx** (const ICplxTrans t)

Description: Transform the text collection with a complex transformation

t: The transformation to apply.

Returns: The transformed texts.

Use of this method is deprecated. Use transformed instead

Transforms the text with the given complex transformation. Does not modify the text collection but returns the transformed texts.

with_match

Signature: *[const]* Texts **with_match** (string pattern, bool inverse)

Description: Filter the text by glob pattern

"pattern" is a glob-style pattern (e.g. "A*" will select all texts starting with a capital "A"). If "inverse" is false, this method returns the texts matching the pattern. If "inverse" is true, this method returns the texts not matching the pattern.

with_text

Signature: *[const]* Texts **with_text** (string text, bool inverse)

Description: Filter the text by text string

If "inverse" is false, this method returns the texts with the given string. If "inverse" is true, this method returns the texts not having the given string.

write

Signature: *[const]* void **write** (string filename)

Description: Writes the region to a file

This method is provided for debugging purposes. It writes the object to a flat layer 0/0 in a single top cell.

This method has been introduced in version 0.29.

4.188. API reference - Class ShapeCollection

Notation used in Ruby API documentation

Module: db

Description: A base class for the shape collections (\Region, \Edges, \EdgePairs and \Texts)

This class has been introduced in version 0.27.

Public constructors

new ShapeCollection ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new ShapeCollection ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

4.189. API reference - Class RdbReference

Notation used in Ruby API documentation

Module: rdb

Description: A cell reference inside the report database

This class describes a cell reference. Such reference object can be attached to cells to describe instantiations of them in parent cells. Not necessarily all instantiations of a cell in the layout database are represented by references and in some cases there might even be no references at all. The references are merely a hint how a marker must be displayed in the context of any other, potentially parent, cell in the layout database.

Public constructors

new RdbReference ptr	**new**	(const DCplxTrans trans, unsigned long parent_cell_id)	Creates a reference with a given transformation and parent cell ID

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const RdbReferenc other)	Assigns another object to self
[const]	const ReportDatabase ptr	**database**		Gets the database object that category is associated with
	ReportDatabase ptr	**database**		Gets the database object that category is associated with (non-const version)
[const]	new RdbReference ptr	**dup**		Creates a copy of self
[const]	unsigned long	**parent_cell_id**		Gets parent cell ID for this reference
	void	**parent_cell_id=**	(unsigned long id)	Sets the parent cell ID for this reference
[const]	DCplxTrans	**trans**		Gets the transformation for this reference
	void	**trans=**	(const DCplxTrans trans)	Sets the transformation for this reference

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const RdbReference other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

database

(1) Signature: *[const]* const ReportDatabase ptr **database**

Description: Gets the database object that category is associated with

This method has been introduced in version 0.23.

(2) Signature: ReportDatabase ptr **database**

Description: Gets the database object that category is associated with (non-const version)

This method has been introduced in version 0.29.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new RdbReference ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new RdbReference ptr **new** (const DCplxTrans trans, unsigned long parent_cell_id)

Description: Creates a reference with a given transformation and parent cell ID

Python specific notes:
This method is the default initializer of the object.

parent_cell_id

Signature: *[const]* unsigned long **parent_cell_id**

Description: Gets parent cell ID for this reference

Returns: The parent cell ID

Python specific notes:
The object exposes a readable attribute 'parent_cell_id'. This is the getter.

parent_cell_id=

Signature: void **parent_cell_id=** (unsigned long id)

Description: Sets the parent cell ID for this reference

Python specific notes:
The object exposes a writable attribute 'parent_cell_id'. This is the setter.

trans

Signature: *[const]* DCplxTrans **trans**

Description: Gets the transformation for this reference

Returns: The transformation

The transformation describes the transformation of the child cell into the parent cell. In that sense that is the usual transformation of a cell reference.

Python specific notes:
The object exposes a readable attribute 'trans'. This is the getter.

trans=

Signature: void **trans=** (const DCplxTrans trans)

Description: Sets the transformation for this reference

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.

4.190. API reference - Class RdbCell

Notation used in Ruby API documentation

Module: rdb

Description: A cell inside the report database

This class represents a cell in the report database. There is not necessarily a 1:1 correspondence of RDB cells and layout database cells. Cells have an ID, a name, optionally a variant name and a set of references which describe at least one example instantiation in some parent cell. The references do not necessarily map to references or cover all references in the layout database.

Public constructors

new RdbCell ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add_reference**	(const RdbRef ref)	Adds a reference to the references of this cell
	void	**clear_references**		Removes all references from this cell
[const]	const ReportDatabase ptr	**database**		Gets the database object that category is associated with
	ReportDatabase ptr	**database**		Gets the database object that category is associated with (non-const version)
[const,iter]	RdbItem	**each_item**		Iterates over all items inside the database which are associated with this cell
[iter]	RdbItem	**each_item**		Iterates over all items inside the database which are associated with this cell (non-const version)
[const,iter]	RdbReference	**each_reference**		Iterates over all references
[iter]	RdbReference	**each_reference**		Iterates over all references (non-const version)
[const]	string	**name**		Gets the cell name
[const]	unsigned long	**num_items**		Gets the number of items for this cell

[const]	unsigned long	**num_items_visited**	Gets the number of visited items for this cell
[const]	string	**qname**	Gets the cell's qualified name
[const]	unsigned long	**rdb_id**	Gets the cell ID
[const]	string	**variant**	Gets the cell variant name

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_reference

Signature: void **add_reference** (const RdbReference ref)

Description: Adds a reference to the references of this cell

ref: The reference to add.

clear_references

Signature: void **clear_references**

Description: Removes all references from this cell

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

database

(1) Signature: *[const]* const ReportDatabase ptr **database**

Description: Gets the database object that category is associated with

This method has been introduced in version 0.23.

(2) Signature: ReportDatabase ptr **database**

Description: Gets the database object that category is associated with (non-const version)

This method has been introduced in version 0.29.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

each_item

(1) Signature: *[const,iter]* RdbItem **each_item**

Description: Iterates over all items inside the database which are associated with this cell

This method has been introduced in version 0.23.

(2) Signature: *[iter]* RdbItem **each_item**

Description: Iterates over all items inside the database which are associated with this cell (non-const version)

This method has been introduced in version 0.29.

each_reference

(1) Signature: *[const,iter]* RdbReference **each_reference**

Description: Iterates over all references

(2) Signature: *[iter]* RdbReference **each_reference**

Description: Iterates over all references (non-const version)

This method has been introduced in version 0.23.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

name

Signature: *[const]* string **name**

Description: Gets the cell name

Returns: The cell name

The cell name is an string that identifies the category in the database. Additionally, a cell may carry a variant identifier which is a string that uniquely identifies a cell in the context of its variants. The "qualified name" contains both the cell name and the variant name. Cell names are also used to identify report database cell's with layout cells.

new

Signature: *[static]* new RdbCell ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

num_items

Signature: *[const]* unsigned long **num_items**

Description: Gets the number of items for this cell

num_items_visited

Signature: *[const]* unsigned long **num_items_visited**

Description: Gets the number of visited items for this cell

qname

Signature: *[const]* string **qname**

Description: Gets the cell's qualified name

Returns: The qualified name

The qualified name is a combination of the cell name and optionally the variant name. It is used to identify the cell by name in a unique way.

rdb_id

Signature: *[const]* unsigned long **rdb_id**

Description: Gets the cell ID

Returns: The cell ID

The cell ID is an integer that uniquely identifies the cell. It is used for referring to a cell in RdbItem for example.

variant

Signature: *[const]* string **variant**

Description: Gets the cell variant name

Returns: The cell variant name

A variant name additionally identifies the cell when multiple cells with the same name are present. A variant name is either assigned automatically or set when creating a cell.

4.191. API reference - Class RdbCategory

Notation used in Ruby API documentation

Module: rdb

Description: A category inside the report database

Every item in the report database is assigned to a category. A category is a DRC rule check for example. Categories can be organized hierarchically, i.e. a category may have sub-categories. Item counts are summarized for categories and items belonging to sub-categories of one category can be browsed together for example. As a general rule, categories not being leaf categories (having child categories) may not have items.

Public constructors

new RdbCategory ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	const ReportDatabase ptr	**database**		Gets the database object that category is associated with
[const]	string	**description**		Gets the category description
	void	**description=**	(string description)	Sets the category description
[const,iter]	RdbItem	**each_item**		Iterates over all items inside the database which are associated with this category
[iter]	RdbItem	**each_item**		Iterates over all items inside the database which are associated with this category (non-const version)
[const,iter]	RdbCategory	**each_sub_category**		Iterates over all sub-categories
[iter]	RdbCategory	**each_sub_category**		Iterates over all sub-categories (non-const version)
[const]	string	**name**		Gets the category name
[const]	unsigned long	**num_items**		Gets the number of items in this category

[const]	unsigned long	**num_items_visited**		Gets the number of visited items in this category
[const]	const RdbCategory ptr	**parent**		Gets the parent category of this category
	RdbCategory ptr	**parent**		Gets the parent category of this category (non-const version)
[const]	string	**path**		Gets the category path
[const]	unsigned long	**rdb_id**		Gets the category ID
	void	**scan_collection**	(RdbCell ptr cell, const CplxTrans trans, const Region region, bool flat = false, bool with_properties = true)	Turns the given region into a hierarchical or flat report database
	void	**scan_collection**	(RdbCell ptr cell, const CplxTrans trans, const Edges edges, bool flat = false, bool with_properties = true)	Turns the given edge collection into a hierarchical or flat report database
	void	**scan_collection**	(RdbCell ptr cell, const CplxTrans trans, const EdgePairs edge_pairs, bool flat = false, bool with_properties = true)	Turns the given edge pair collection into a hierarchical or flat report database
	void	**scan_collection**	(RdbCell ptr cell, const CplxTrans trans, const Texts texts, bool flat = false, bool with_properties = true)	Turns the given edge pair collection into a hierarchical or flat report database
	void	**scan_layer**	(const Layout layout, unsigned int layer, const Cell ptr cell = nil, int levels = -1,	Scans a layer from a layout into this category, starting with a given cell and a depth specification

			bool with_properties = true)	
	void	scan_shapes	(const RecursiveShapeIterator iter, bool flat = false, bool with_properties = true)	Scans the polygon or edge shapes from the shape iterator into the category

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

database

Signature: *[const]* const ReportDatabase ptr **database**

Description: Gets the database object that category is associated with

This method has been introduced in version 0.23.

description

Signature: *[const]* string **description**

Description: Gets the category description

Returns: The description string

Python specific notes:
The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string description)

Description: Sets the category description

description: The description string

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

each_item

(1) Signature: *[const,iter]* RdbItem **each_item**

Description: Iterates over all items inside the database which are associated with this category

This method has been introduced in version 0.23.

(2) Signature: *[iter]* RdbItem **each_item**

Description: Iterates over all items inside the database which are associated with this category (non-const version)

This method has been introduced in version 0.29.

each_sub_category

(1) Signature: *[const,iter]* RdbCategory **each_sub_category**

Description: Iterates over all sub-categories

The const version has been added in version 0.29.

(2) Signature: *[iter]* RdbCategory **each_sub_category**

Description: Iterates over all sub-categories (non-const version)

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

name

Signature: *[const]* string **name**

Description: Gets the category name

Returns: The category name

The category name is an string that identifies the category in the context of a parent category or inside the database when it is a top level category. The name is not the path name which is a path to a child category and incorporates all names of parent categories.

new

Signature: *[static]* new RdbCategory ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

num_items

Signature: *[const]* unsigned long **num_items**

Description: Gets the number of items in this category

The number of items includes the items in sub-categories of this category.

num_items_visited

Signature: *[const]* unsigned long **num_items_visited**

Description: Gets the number of visited items in this category

The number of items includes the items in sub-categories of this category.

parent

(1) Signature: *[const]* const RdbCategory ptr **parent**

Description: Gets the parent category of this category

Returns: The parent category or nil if this category is a top-level category

The const version has been added in version 0.29.

(2) Signature: RdbCategory ptr **parent**

Description: Gets the parent category of this category (non-const version)

Returns: The parent category or nil if this category is a top-level category

path

Signature: *[const]* string **path**

Description: Gets the category path

Returns: The path for this category

The category path is the category name for top level categories. For child categories, the path contains the names of all parent categories separated by a dot.

rdb_id

Signature: *[const]* unsigned long **rdb_id**

Description: Gets the category ID

Returns: The category ID

The category ID is an integer that uniquely identifies the category. It is used for referring to a category in RdbItem for example.

scan_collection

(1) Signature: void **scan_collection** (RdbCell ptr cell, const CplxTrans trans, const Region region, bool flat = false, bool with_properties = true)

Description: Turns the given region into a hierarchical or flat report database

The exact behavior depends on the nature of the region. If the region is a hierarchical (original or deep) region and the 'flat' argument is false, this method will produce a hierarchical report database in the given category. The 'cell_id' parameter is ignored in this case. Sample references will be produced to supply minimal instantiation information.

If the region is a flat one or the 'flat' argument is true, the region's polygons will be produced as report database items in this category and in the cell given by 'cell_id'.

The transformation argument needs to supply the dbu-to-micron transformation.

If 'with_properties' is true, user properties will be turned into tagged values as well.

This method has been introduced in version 0.26. The 'with_properties' argument has been added in version 0.28.

(2) Signature: void **scan_collection** (RdbCell ptr cell, const CplxTrans trans, const Edges edges, bool flat = false, bool with_properties = true)

Description: Turns the given edge collection into a hierarchical or flat report database

This a another flavour of scan_collection accepting an edge collection.

This method has been introduced in version 0.26. The 'with_properties' argument has been added in version 0.28.

(3) Signature: void **scan_collection** (RdbCell ptr cell, const CplxTrans trans, const EdgePairs edge_pairs, bool flat = false, bool with_properties = true)

Description: Turns the given edge pair collection into a hierarchical or flat report database

This a another flavour of scan_collection accepting an edge pair collection.

This method has been introduced in version 0.26. The 'with_properties' argument has been added in version 0.28.

(4) Signature: void **scan_collection** (RdbCell ptr cell, const CplxTrans trans, const Texts texts, bool flat = false, bool with_properties = true)

Description: Turns the given edge pair collection into a hierarchical or flat report database

This a another flavour of scan_collection accepting a text collection.

This method has been introduced in version 0.28.

scan_layer

Signature: void **scan_layer** (const Layout layout, unsigned int layer, const Cell ptr cell = nil, int levels = -1, bool with_properties = true)

Description: Scans a layer from a layout into this category, starting with a given cell and a depth specification

Creates RDB items for each polygon or edge shape read from the cell and its children in the layout on the given layer and puts them into this category. New cells will be generated when required. "levels" is the number of hierarchy levels to take the child cells from. 0 means to use only "cell" and don't descend, -1 means "all levels". Other settings like database unit, description, top cell etc. are not made in the RDB.

If 'with_properties' is true, user properties will be turned into tagged values as well.

This method has been introduced in version 0.23. The 'with_properties' argument has been added in version 0.28.

scan_shapes

Signature: void **scan_shapes** (const RecursiveShapeIterator iter, bool flat = false, bool with_properties = true)

Description: Scans the polygon or edge shapes from the shape iterator into the category

Creates RDB items for each polygon or edge shape read from the iterator and puts them into this category. A similar, but lower-level method is ReportDatabase#create_items with a RecursiveShapeIterator argument. In contrast to ReportDatabase#create_items, 'scan_shapes' can also produce hierarchical databases if the flat argument is false. In this case, the hierarchy the recursive shape iterator traverses is copied into the report database using sample references.

If 'with_properties' is true, user properties will be turned into tagged values as well.

This method has been introduced in version 0.23. The flat mode argument has been added in version 0.26. The 'with_properties' argument has been added in version 0.28.

4.192. API reference - Class RdbItemValue

Notation used in Ruby API documentation

Module: rdb

Description: A value object inside the report database

Value objects are attached to items to provide markers. An arbitrary number of such value objects can be attached to an item. Currently, a value can represent a box, a polygon or an edge. Geometrical objects are represented in micron units and are therefore "D" type objects (DPolygon, DEdge and DBox).

Public constructors

new RdbItemValue ptr	**new**	(double f)	Creates a value representing a numeric value
new RdbItemValue ptr	**new**	(string s)	Creates a value representing a string
new RdbItemValue ptr	**new**	(const DPolygon p)	Creates a value representing a DPolygon object
new RdbItemValue ptr	**new**	(const DPath p)	Creates a value representing a DPath object
new RdbItemValue ptr	**new**	(const DText t)	Creates a value representing a DText object
new RdbItemValue ptr	**new**	(const DEdge e)	Creates a value representing a DEdge object
new RdbItemValue ptr	**new**	(const DEdgePair ee)	Creates a value representing a DEdgePair object
new RdbItemValue ptr	**new**	(const DBox b)	Creates a value representing a DBox object

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const RdbItem other)	Assigns another object to self
[const]	DBox	**box**		Gets the box if the value represents one.
[const]	new RdbItemValue ptr	**dup**		Creates a copy of self

[const]	DEdge	**edge**		Gets the edge if the value represents one.
[const]	DEdgePair	**edge_pair**		Gets the edge pair if the value represents one.
[const]	double	**float**		Gets the numeric value.
[const]	bool	**is_box?**		Returns true if the value object represents a box
[const]	bool	**is_edge?**		Returns true if the value object represents an edge
[const]	bool	**is_edge_pair?**		Returns true if the value object represents an edge pair
[const]	bool	**is_float?**		Returns true if the value object represents a numeric value
[const]	bool	**is_path?**		Returns true if the value object represents a path
[const]	bool	**is_polygon?**		Returns true if the value object represents a polygon
[const]	bool	**is_string?**		Returns true if the object represents a string value
[const]	bool	**is_text?**		Returns true if the value object represents a text
[const]	DPath	**path**		Gets the path if the value represents one.
[const]	DPolygon	**polygon**		Gets the polygon if the value represents one.
[const]	string	**string**		Gets the string representation of the value.
[const]	unsigned long	**tag_id**		Gets the tag ID if the value is a tagged value or 0 if not
	void	**tag_id=**	(unsigned long id)	Sets the tag ID to make the value a tagged value or 0 to reset it
[const]	DText	**text**		Gets the text if the value represents one.
[const]	string	**to_s**		Converts a value to a string

Public static methods and constants

RdbItemValue ptr	**from_s**	(string s)	Creates a value object from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
_destroy	**Signature**: void **_destroy** **Description**: Explicitly destroys the object Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
_destroyed?	**Signature**: *[const]* bool **_destroyed?** **Description**: Returns a value indicating whether the object was already destroyed This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
_is_const_object?	**Signature**: *[const]* bool **_is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference This method returns true, if self is a const reference. In that case, only const methods may be called on self.
_manage	**Signature**: void **_manage** **Description**: Marks the object as managed by the script side. After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
assign	**Signature**: void **assign** (const RdbItemValue other) **Description**: Assigns another object to self
box	**Signature**: *[const]* DBox **box** **Description**: Gets the box if the value represents one.

Returns: The DBox object or nil

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new RdbItemValue ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

edge

Signature: *[const]* DEdge **edge**

Description: Gets the edge if the value represents one.

Returns: The DEdge object or nil

edge_pair

Signature: *[const]* DEdgePair **edge_pair**

Description: Gets the edge pair if the value represents one.

Returns: The DEdgePair object or nil

float

Signature: *[const]* double **float**

Description: Gets the numeric value.

Returns: The numeric value or 0

This method has been introduced in version 0.24.

from_s

Signature: *[static]* RdbItemValue ptr **from_s** (string s)

Description: Creates a value object from a string

The string format is the same than obtained by the to_s method.

is_box?	**Signature**: *[const]* bool **is_box?** **Description**: Returns true if the value object represents a box
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
is_edge?	**Signature**: *[const]* bool **is_edge?** **Description**: Returns true if the value object represents an edge
is_edge_pair?	**Signature**: *[const]* bool **is_edge_pair?** **Description**: Returns true if the value object represents an edge pair
is_float?	**Signature**: *[const]* bool **is_float?** **Description**: Returns true if the value object represents a numeric value This method has been introduced in version 0.24.
is_path?	**Signature**: *[const]* bool **is_path?** **Description**: Returns true if the value object represents a path This method has been introduced in version 0.22.
is_polygon?	**Signature**: *[const]* bool **is_polygon?** **Description**: Returns true if the value object represents a polygon
is_string?	**Signature**: *[const]* bool **is_string?** **Description**: Returns true if the object represents a string value
is_text?	**Signature**: *[const]* bool **is_text?** **Description**: Returns true if the value object represents a text This method has been introduced in version 0.22.
new	**(1) Signature**: *[static]* new RdbItemValue ptr **new** (double f) **Description**: Creates a value representing a numeric value This variant has been introduced in version 0.24 **Python specific notes:** This method is the default initializer of the object. **(2) Signature**: *[static]* new RdbItemValue ptr **new** (string s) **Description**: Creates a value representing a string **Python specific notes:** This method is the default initializer of the object.

(3) Signature: *[static]* new RdbItemValue ptr **new** (const DPolygon p)

Description: Creates a value representing a DPolygon object

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new RdbItemValue ptr **new** (const DPath p)

Description: Creates a value representing a DPath object

This method has been introduced in version 0.22.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new RdbItemValue ptr **new** (const DText t)

Description: Creates a value representing a DText object

This method has been introduced in version 0.22.

Python specific notes:
This method is the default initializer of the object.

(6) Signature: *[static]* new RdbItemValue ptr **new** (const DEdge e)

Description: Creates a value representing a DEdge object

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new RdbItemValue ptr **new** (const DEdgePair ee)

Description: Creates a value representing a DEdgePair object

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new RdbItemValue ptr **new** (const DBox b)

Description: Creates a value representing a DBox object

Python specific notes:
This method is the default initializer of the object.

path

Signature: *[const]* DPath **path**

Description: Gets the path if the value represents one.

Returns: The DPath object

This method has been introduced in version 0.22.

polygon

Signature: *[const]* DPolygon **polygon**

Description: Gets the polygon if the value represents one.

Returns: The DPolygon object

string

Signature: *[const]* string **string**

Description: Gets the string representation of the value.

Returns: The stringThis method will always deliver a valid string, even if is_string? is false. The objects stored in the value are converted to a string accordingly.

tag_id

Signature: *[const]* unsigned long **tag_id**

Description: Gets the tag ID if the value is a tagged value or 0 if not

Returns: The tag ID

See tag_id= for details about tagged values.

Tagged values have been added in version 0.24.

Python specific notes:
The object exposes a readable attribute 'tag_id'. This is the getter.

tag_id=

Signature: void **tag_id=** (unsigned long id)

Description: Sets the tag ID to make the value a tagged value or 0 to reset it

id: The tag ID

To get a tag ID, use RdbDatabase#user_tag_id (preferred) or RdbDatabase#tag_id (for internal use). Tagged values have been added in version 0.24. Tags can be given to identify a value, for example to attache measurement values to an item. To attach a value for a specific measurement, a tagged value can be used where the tag ID describes the measurement made. In that way, multiple values for different measurements can be attached to an item.

This variant has been introduced in version 0.24

Python specific notes:
The object exposes a writable attribute 'tag_id'. This is the setter.

text

Signature: *[const]* DText **text**

Description: Gets the text if the value represents one.

Returns: The DText object

This method has been introduced in version 0.22.

to_s

Signature: *[const]* string **to_s**

Description: Converts a value to a string

Returns: The string

The string can be used by the string constructor to create another object from it.

Python specific notes:
This method is also available as 'str(object)'.

4.193. API reference - Class RdbItem

Notation used in Ruby API documentation

Module: rdb

Description: An item inside the report database

An item is the basic information entity in the RDB. It is associated with a cell and a category. It can be assigned values which encapsulate other objects such as strings and geometrical objects. In addition, items can be assigned an image (i.e. a screenshot image) and tags which are basically boolean flags that can be defined freely.

Public constructors

new RdbItem ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add_tag**	(unsigned long tag_id)	Adds a tag with the given id to the item
	void	**add_value**	(const RdbItemValue value)	Adds a value object to the values of this item
	void	**add_value**	(const DPolygon value)	Adds a polygon object to the values of this item
	void	**add_value**	(const DBox value)	Adds a box object to the values of this item
	void	**add_value**	(const DEdge value)	Adds an edge object to the values of this item
	void	**add_value**	(const DEdgePair value)	Adds an edge pair object to the values of this item
	void	**add_value**	(string value)	Adds a string object to the values of this item
	void	**add_value**	(double value)	Adds a numeric value to the values of this item

	void	add_value	(const Shape shape, const CplxTrans trans)	Adds a geometrical value object from a shape
	void	assign	(const RdbItem other)	Assigns another object to self
[const]	unsigned long	category_id		Gets the category ID
[const]	unsigned long	cell_id		Gets the cell ID
	void	clear_values		Removes all values from this item
[const]	const ReportDatabase ptr	database		Gets the database object that item is associated with
[const]	new RdbItem ptr	dup		Creates a copy of self
[const,iter]	RdbItemValue	each_value		Iterates over all values
[const]	bool	has_image?		Gets a value indicating that the item has an image attached
[const]	bool	has_tag?	(unsigned long tag_id)	Returns a value indicating whether the item has a tag with the given ID
[const]	QImage	image		Gets the attached image as a QImage object
	void	image=	(const QImage arg1)	Sets the attached image from a QImage object
[const]	string	image_str		Gets the image associated with this item as a string
	void	image_str=	(string image)	Sets the image from a string
[const]	bool	is_visited?		Gets a value indicating whether the item was already visited
	void	remove_tag	(unsigned long tag_id)	Remove the tag with the given id from the item
[const]	string	tags_str		Returns a string listing all tags of this item
	void	tags_str=	(string tags)	Sets the tags from a string

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead

[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_tag

Signature: void **add_tag** (unsigned long tag_id)

Description: Adds a tag with the given id to the item

Each tag can be added once to the item. The tags of an item thus form a set. If a tag with that ID already exists, this method does nothing.

add_value

(1) Signature: void **add_value** (const RdbItemValue value)

Description: Adds a value object to the values of this item

value: The value to add.

(2) Signature: void **add_value** (const DPolygon value)

Description: Adds a polygon object to the values of this item

value: The polygon to add.

This method has been introduced in version 0.25 as a convenience method.

(3) Signature: void **add_value** (const DBox value)

Description: Adds a box object to the values of this item

value: The box to add.

This method has been introduced in version 0.25 as a convenience method.

(4) Signature: void **add_value** (const DEdge value)

Description: Adds an edge object to the values of this item

value: The edge to add.

This method has been introduced in version 0.25 as a convenience method.

(5) Signature: void **add_value** (const DEdgePair value)

Description: Adds an edge pair object to the values of this item

value: The edge pair to add.

This method has been introduced in version 0.25 as a convenience method.

(6) Signature: void **add_value** (string value)

Description: Adds a string object to the values of this item

value: The string to add.

This method has been introduced in version 0.25 as a convenience method.

(7) Signature: void **add_value** (double value)

Description: Adds a numeric value to the values of this item

value: The value to add.

This method has been introduced in version 0.25 as a convenience method.

(8) Signature: void **add_value** (const Shape shape, const CplxTrans trans)

Description: Adds a geometrical value object from a shape

value: The shape object from which to take the geometrical object.

trans: The transformation to apply.

The transformation can be used to convert database units to micron units.

This method has been introduced in version 0.25.3.

assign

Signature: void **assign** (const RdbItem other)

Description: Assigns another object to self

category_id

Signature: *[const]* unsigned long **category_id**

Description: Gets the category ID

Returns: The category ID

Returns the ID of the category that this item is associated with.

cell_id

Signature: *[const]* unsigned long **cell_id**

Description: Gets the cell ID

Returns: The cell ID

Returns the ID of the cell that this item is associated with.

clear_values

Signature: void **clear_values**

Description: Removes all values from this item

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

database

Signature: *[const]* const ReportDatabase ptr **database**

Description: Gets the database object that item is associated with

This method has been introduced in version 0.23.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new RdbItem ptr **dup**

Description: Creates a copy of self

Python specific notes:

This method also implements '__copy__' and '__deepcopy__'.

each_value

Signature: *[const,iter]* RdbItemValue **each_value**

Description: Iterates over all values

has_image?

Signature: *[const]* bool **has_image?**

Description: Gets a value indicating that the item has an image attached

See image_str how to obtain the image.

This method has been introduced in version 0.28.

has_tag?

Signature: *[const]* bool **has_tag?** (unsigned long tag_id)

Description: Returns a value indicating whether the item has a tag with the given ID

Returns: True, if the item has a tag with the given ID

image

Signature: *[const]* QImage **image**

Description: Gets the attached image as a QImage object

This method has been added in version 0.28.

Python specific notes:
The object exposes a readable attribute 'image'. This is the getter.

image=

Signature: void **image=** (const QImage arg1)

Description: Sets the attached image from a QImage object

This method has been added in version 0.28.

Python specific notes:
The object exposes a writable attribute 'image'. This is the setter.

image_str

Signature: *[const]* string **image_str**

Description: Gets the image associated with this item as a string

Returns: A base64-encoded image file (in PNG format)

Python specific notes:
The object exposes a readable attribute 'image_str'. This is the getter.

image_str=

Signature: void **image_str=** (string image)

Description: Sets the image from a string

image: A base64-encoded image file (preferably in PNG format)

Python specific notes:
The object exposes a writable attribute 'image_str'. This is the setter.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_visited?

Signature: *[const]* bool **is_visited?**

Description: Gets a value indicating whether the item was already visited

Returns: True, if the item has been visited already

new

Signature: *[static]* new RdbItem ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

remove_tag

Signature: void **remove_tag** (unsigned long tag_id)

Description: Remove the tag with the given id from the item

If a tag with that ID does not exists on this item, this method does nothing.

tags_str

Signature: *[const]* string **tags_str**

Description: Returns a string listing all tags of this item

Returns: A comma-separated list of tags

Python specific notes:
The object exposes a readable attribute 'tags_str'. This is the getter.

tags_str=

Signature: void **tags_str=** (string tags)

Description: Sets the tags from a string

tags: A comma-separated list of tags

Python specific notes:
The object exposes a writable attribute 'tags_str'. This is the setter.

4.194. API reference - Class ReportDatabase

Notation used in Ruby API documentation

Module: rdb

Description: The report database object

A report database is organized around a set of items which are associated with cells and categories. Categories can be organized hierarchically by created sub-categories of other categories. Cells are associated with layout database cells and can come with a example instantiation if the layout database does not allow a unique association of the cells. Items in the database can have a variety of attributes: values, tags and an image object. Values are geometrical objects for example. Tags are a set of boolean flags and an image can be attached to an item to provide a screenshot for visualization for example. This is the main report database object. The basic use case of this object is to create one inside a LayoutView and populate it with items, cell and categories or load it from a file. Another use case is to create a standalone ReportDatabase object and use the methods provided to perform queries or to populate it.

Public constructors

new ReportDatabase ptr	**new**	(string name)	Creates a report database

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	const RdbCategory ptr	**category_by_id**	(unsigned long id)	Gets a category by ID
	RdbCategory ptr	**category_by_id**	(unsigned long id)	Gets a category by ID (non-const version)
[const]	const RdbCategory ptr	**category_by_path**	(string path)	Gets a category by path
	RdbCategory ptr	**category_by_path**	(string path)	Gets a category by path (non-const version)
[const]	const RdbCell ptr	**cell_by_id**	(unsigned long id)	Returns the cell for a given ID
	RdbCell ptr	**cell_by_id**	(unsigned long id)	Returns the cell for a given ID (non-const version)
[const]	const RdbCell ptr	**cell_by_qname**	(string qname)	Returns the cell for a given qualified name

	RdbCell ptr	**cell_by_qname**	(string qname)	Returns the cell for a given qualified name (non-const version)
	RdbCategory ptr	**create_category**	(string name)	Creates a new top level category
	RdbCategory ptr	**create_category**	(RdbCategory ptr parent, string name)	Creates a new sub-category
	RdbCell ptr	**create_cell**	(string name)	Creates a new cell
	RdbCell ptr	**create_cell**	(string name, string variant)	Creates a new cell, potentially as a variant for a cell with the same name
	RdbItem ptr	**create_item**	(unsigned long cell_id, unsigned long category_id)	Creates a new item for the given cell/category combination
	RdbItem ptr	**create_item**	(RdbCell ptr cell, RdbCategory ptr category)	Creates a new item for the given cell/category combination
	void	**create_item**	(unsigned long cell_id, unsigned long category_id, const CplxTrans trans, const Shape shape, bool with_properties = true)	Creates a new item from a single shape
	void	**create_items**	(unsigned long cell_id, unsigned long category_id, const RecursiveShapeIterator iter, bool with_properties = true)	Creates new items from a shape iterator
	void	**create_items**	(unsigned long cell_id, unsigned long category_id, const CplxTrans trans, const Shapes shapes, bool with_properties = true)	Creates new items from a shape container
	void	**create_items**	(unsigned long cell_id, unsigned long category_id, const CplxTrans trans, Polygon[] array)	Creates new polygon items for the given cell/category combination
	void	**create_items**	(unsigned long cell_id, unsigned long category_id, const CplxTrans trans, Edge[] array)	Creates new edge items for the given cell/category combination

	void	**create_items**	(unsigned long cell_id, unsigned long category_id, const CplxTrans trans, EdgePair[] array)	Creates new edge pair items for the given cell/category combination
[const]	string	**description**		Gets the databases description
	void	**description=**	(string desc)	Sets the databases description
[const,iter]	RdbCategory	**each_category**		Iterates over all top-level categories
[iter]	RdbCategory	**each_category**		Iterates over all top-level categories (non-const version)
[const,iter]	RdbCell	**each_cell**		Iterates over all cells
[iter]	RdbCell	**each_cell**		Iterates over all cells (non-const version)
[const,iter]	RdbItem	**each_item**		Iterates over all items inside the database
[iter]	RdbItem	**each_item**		Iterates over all items inside the database (non-const version)
[const,iter]	RdbItem	**each_item_per_catego**	(unsigned long category_id)	Iterates over all items inside the database which are associated with the given category
[iter]	RdbItem	**each_item_per_category**	(unsigned long category_id)	Iterates over all items inside the database which are associated with the given category (non-const version)
[const,iter]	RdbItem	**each_item_per_cell**	(unsigned long cell_id)	Iterates over all items inside the database which are associated with the given cell
[iter]	RdbItem	**each_item_per_cell**	(unsigned long cell_id)	Iterates over all items inside the database which are associated with the given cell (non-const version)
[const,iter]	RdbItem	**each_item_per_cell_an**	(unsigned long cell_id, unsigned long category_id)	Iterates over all items inside the database which are associated with the given cell and category
[iter]	RdbItem	**each_item_per_cell_and**	(unsigned long cell_id, unsigned long category_id)	Iterates over all items inside the database which are associated with the given cell and category
[const]	string	**filename**		Gets the file name and path where the report database is stored
[const]	string	**generator**		Gets the databases generator
	void	**generator=**	(string generator)	Sets the generator string
[const]	bool	**is_modified?**		Returns a value indicating whether the database has been modified
	void	**load**	(string filename)	Loads the database from the given file

[const]	string	**name**		Gets the database name
[const]	unsigned long	**num_items**		Returns the number of items inside the database
[const]	unsigned long	**num_items**	(unsigned long cell_id, unsigned long category_id)	Returns the number of items inside the database for a given cell/category combination
[const]	unsigned long	**num_items_visited**		Returns the number of items already visited inside the database
[const]	unsigned long	**num_items_visited**	(unsigned long cell_id, unsigned long category_id)	Returns the number of items visited already for a given cell/category combination
[const]	string	**original_file**		Gets the original file name and path
	void	**original_file=**	(string path)	Sets the original file name and path
	void	**reset_modified**		Reset the modified flag
	void	**save**	(string filename)	Saves the database to the given file
	void	**set_item_visited**	(const RdbItem ptr item, bool visited)	Modifies the visited state of an item
	void	**set_tag_description**	(unsigned long tag_id, string description)	Sets the tag description for the given tag ID
[const]	string	**tag_description**	(unsigned long tag_id)	Gets the tag description for the given tag ID
[const]	unsigned long	**tag_id**	(string name)	Gets the tag ID for a given tag name
[const]	string	**tag_name**	(unsigned long tag_id)	Gets the tag name for the given tag ID
[const]	string	**top_cell_name**		Gets the top cell name
	void	**top_cell_name=**	(string cell_name)	Sets the top cell name string
[const]	unsigned long	**user_tag_id**	(string name)	Gets the tag ID for a given user tag name
	unsigned long[]	**variants**	(string name)	Gets the variants for a given cell name

Deprecated methods (protected, public, static, non-static and constructors)

void	**create**		Use of this method is deprecated. Use _create instead
void	**create_items**	(unsigned long cell_id, unsigned long category_id, const CplxTrans trans, const Region region)	Use of this method is deprecated
void	**create_items**	(unsigned long cell_id, unsigned long category_id,	Use of this method is deprecated

			const CplxTrans trans, const Edges edges)	
	void	create_items	(unsigned long cell_id, unsigned long category_id, const CplxTrans trans, const EdgePairs edge_pairs)	Use of this method is deprecated
	void	destroy		Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

category_by_id

(1) Signature: *[const]* const RdbCategory ptr **category_by_id** (unsigned long id)

Description: Gets a category by ID

Returns: The (const) category object or nil if the ID is not valid

(2) Signature: RdbCategory ptr **category_by_id** (unsigned long id)

Description: Gets a category by ID (non-const version)

Returns: The (const) category object or nil if the ID is not valid

This non-const variant has been introduced in version 0.29.

category_by_path

(1) Signature: *[const]* const RdbCategory ptr **category_by_path** (string path)

Description: Gets a category by path

path: The full path to the category starting from the top level (subcategories separated by dots)

Returns: The (const) category object or nil if the name is not valid

(2) Signature: RdbCategory ptr **category_by_path** (string path)

Description: Gets a category by path (non-const version)

path: The full path to the category starting from the top level (subcategories separated by dots)

Returns: The (const) category object or nil if the name is not valid

This non-const variant has been introduced in version 0.29.

cell_by_id

(1) Signature: *[const]* const RdbCell ptr **cell_by_id** (unsigned long id)

Description: Returns the cell for a given ID

id: The ID of the cell

Returns: The cell object or nil if no cell with that ID exists

(2) Signature: RdbCell ptr **cell_by_id** (unsigned long id)

Description: Returns the cell for a given ID (non-const version)

id: The ID of the cell

Returns: The cell object or nil if no cell with that ID exists

This non-const variant has been added version 0.29.

cell_by_qname

(1) Signature: *[const]* const RdbCell ptr **cell_by_qname** (string qname)

Description: Returns the cell for a given qualified name

qname:	The qualified name of the cell (name plus variant name optionally)
Returns:	The cell object or nil if no such cell exists

(2) Signature: RdbCell ptr **cell_by_qname** (string qname)

Description: Returns the cell for a given qualified name (non-const version)

qname:	The qualified name of the cell (name plus variant name optionally)
Returns:	The cell object or nil if no such cell exists

This non-const variant has been added version 0.29.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

create_category

(1) Signature: RdbCategory ptr **create_category** (string name)

Description: Creates a new top level category

name:	The name of the category

(2) Signature: RdbCategory ptr **create_category** (RdbCategory ptr parent, string name)

Description: Creates a new sub-category

parent:	The category under which the category should be created
name:	The name of the category

create_cell

(1) Signature: RdbCell ptr **create_cell** (string name)

Description: Creates a new cell

name:	The name of the cell

(2) Signature: RdbCell ptr **create_cell** (string name, string variant)

Description: Creates a new cell, potentially as a variant for a cell with the same name

name:	The name of the cell
variant:	The variant name of the cell

create_item

(1) Signature: RdbItem ptr **create_item** (unsigned long cell_id, unsigned long category_id)

Description: Creates a new item for the given cell/category combination

cell_id:	The ID of the cell to which the item is associated
category_id:	The ID of the category to which the item is associated

A more convenient method that takes cell and category objects instead of ID's is the other version of create_item.

(2) Signature: RdbItem ptr **create_item** (RdbCell ptr cell, RdbCategory ptr category)

Description: Creates a new item for the given cell/category combination

cell:	The cell to which the item is associated
category:	The category to which the item is associated

This convenience method has been added in version 0.25.

(3) Signature: void **create_item** (unsigned long cell_id, unsigned long category_id, const CplxTrans trans, const Shape shape, bool with_properties = true)

Description: Creates a new item from a single shape

cell_id:	The ID of the cell to which the item is associated
category_id:	The ID of the category to which the item is associated
shape:	The shape to take the geometrical object from
trans:	The transformation to apply
with_properties:	If true, user properties will be turned into tagged values as well

This method produces an item from the given shape. It accepts various kind of shapes, such as texts, polygons, boxes and paths and converts them to a corresponding item. The transformation argument can be used to supply the transformation that applies the database unit for example.

This method has been introduced in version 0.25.3. The 'with_properties' argument has been added in version 0.28.

create_items

(1) Signature: void **create_items** (unsigned long cell_id, unsigned long category_id, const RecursiveShapeIterator iter, bool with_properties = true)

Description: Creates new items from a shape iterator

cell_id:	The ID of the cell to which the item is associated
category_id:	The ID of the category to which the item is associated
iter:	The iterator (a RecursiveShapeIterator object) from which to take the items
with_properties:	If true, user properties will be turned into tagged values as well

This method takes the shapes from the given iterator and produces items from them. It accepts various kind of shapes, such as texts, polygons, boxes and paths and converts them to corresponding items. This method will produce a flat version of the shapes iterated by the shape iterator. A similar method, which is intended for production of polygon or edge error layers and also provides hierarchical database construction is RdbCategory#scan_shapes.

This method has been introduced in version 0.25.3. The 'with_properties' argument has been added in version 0.28.

(2) Signature: void **create_items** (unsigned long cell_id, unsigned long category_id, const CplxTrans trans, const Shapes shapes, bool with_properties = true)

Description: Creates new items from a shape container

cell_id:	The ID of the cell to which the item is associated
category_id:	The ID of the category to which the item is associated
shapes:	The shape container from which to take the items

trans:	The transformation to apply
with_properties:	If true, user properties will be turned into tagged values as well

This method takes the shapes from the given container and produces items from them. It accepts various kind of shapes, such as texts, polygons, boxes and paths and converts them to corresponding items. The transformation argument can be used to supply the transformation that applies the database unit for example.

This method has been introduced in version 0.25.3. The 'with_properties' argument has been added in version 0.28.

(3) Signature: void **create_items** (unsigned long cell_id, unsigned long category_id, const CplxTrans trans, const Region region)

Description: Creates new polygon items for the given cell/category combination

cell_id:	The ID of the cell to which the item is associated
category_id:	The ID of the category to which the item is associated
trans:	The transformation to apply
region:	The region (a Region object) containing the polygons for which to create items

Use of this method is deprecated

For each polygon in the region a single item will be created. The value of the item will be this polygon. A transformation can be supplied which can be used for example to convert the object's dimensions to micron units by scaling by the database unit.

This method will also produce a flat version of the shapes inside the region. RdbCategory#scan_collection is a similar method which also supports construction of hierarchical databases from deep regions.

This method has been introduced in version 0.23. It has been deprecated in favor of RdbCategory#scan_collection in version 0.28.

(4) Signature: void **create_items** (unsigned long cell_id, unsigned long category_id, const CplxTrans trans, const Edges edges)

Description: Creates new edge items for the given cell/category combination

cell_id:	The ID of the cell to which the item is associated
category_id:	The ID of the category to which the item is associated
trans:	The transformation to apply
edges:	The list of edges (an Edges object) for which the items are created

Use of this method is deprecated

For each edge a single item will be created. The value of the item will be this edge. A transformation can be supplied which can be used for example to convert the object's dimensions to micron units by scaling by the database unit.

This method will also produce a flat version of the edges inside the edge collection. RdbCategory#scan_collection is a similar method which also supports construction of hierarchical databases from deep edge collections.

This method has been introduced in version 0.23. It has been deprecated in favor of RdbCategory#scan_collection in version 0.28.

(5) Signature: void **create_items** (unsigned long cell_id, unsigned long category_id, const CplxTrans trans, const EdgePairs edge_pairs)

Description: Creates new edge pair items for the given cell/category combination

cell_id:	The ID of the cell to which the item is associated
category_id:	The ID of the category to which the item is associated
trans:	The transformation to apply
edges:	The list of edge pairs (an EdgePairs object) for which the items are created

Use of this method is deprecated

For each edge pair a single item will be created. The value of the item will be this edge pair. A transformation can be supplied which can be used for example to convert the object's dimensions to micron units by scaling by the database unit.

This method will also produce a flat version of the edge pairs inside the edge pair collection. RdbCategory#scan_collection is a similar method which also supports construction of hierarchical databases from deep edge pair collections.

This method has been introduced in version 0.23. It has been deprecated in favor of RdbCategory#scan_collection in version 0.28.

(6) Signature: void **create_items** (unsigned long cell_id, unsigned long category_id, const CplxTrans trans, Polygon[] array)

Description: Creates new polygon items for the given cell/category combination

cell_id:	The ID of the cell to which the item is associated
category_id:	The ID of the category to which the item is associated
trans:	The transformation to apply
polygons:	The list of polygons for which the items are created

For each polygon a single item will be created. The value of the item will be this polygon. A transformation can be supplied which can be used for example to convert the object's dimensions to micron units by scaling by the database unit.

This method has been introduced in version 0.23.

(7) Signature: void **create_items** (unsigned long cell_id, unsigned long category_id, const CplxTrans trans, Edge[] array)

Description: Creates new edge items for the given cell/category combination

cell_id:	The ID of the cell to which the item is associated
category_id:	The ID of the category to which the item is associated
trans:	The transformation to apply
edges:	The list of edges for which the items are created

For each edge a single item will be created. The value of the item will be this edge. A transformation can be supplied which can be used for example to convert the object's dimensions to micron units by scaling by the database unit.

This method has been introduced in version 0.23.

(8) Signature: void **create_items** (unsigned long cell_id, unsigned long category_id, const CplxTrans trans, EdgePair[] array)

Description: Creates new edge pair items for the given cell/category combination

cell_id:	The ID of the cell to which the item is associated
category_id:	The ID of the category to which the item is associated
trans:	The transformation to apply
edge_pairs:	The list of edge_pairs for which the items are created

For each edge pair a single item will be created. The value of the item will be this edge pair. A transformation can be supplied which can be used for example to convert the object's dimensions to micron units by scaling by the database unit.

This method has been introduced in version 0.23.

description

Signature: *[const]* string **description**

Description: Gets the databases description

Returns: The description string

The description is a general purpose string that is supposed to further describe the database and its content in a human-readable form.

Python specific notes:
The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string desc)

Description: Sets the databases description

desc: The description string

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

each_category

(1) Signature: *[const,iter]* RdbCategory **each_category**

Description: Iterates over all top-level categories

(2) Signature: *[iter]* RdbCategory **each_category**

Description: Iterates over all top-level categories (non-const version)

The non-const variant has been added in version 0.29.

each_cell

(1) Signature: *[const,iter]* RdbCell **each_cell**

Description: Iterates over all cells

(2) Signature: *[iter]* RdbCell **each_cell**

Description: Iterates over all cells (non-const version)

This non-const variant has been added version 0.29.

each_item

(1) Signature: *[const,iter]* RdbItem **each_item**

Description: Iterates over all items inside the database

(2) Signature: *[iter]* RdbItem **each_item**

Description: Iterates over all items inside the database (non-const version)

This non-const variant has been added in version 0.29.

each_item_per_category

(1) Signature: *[const,iter]* RdbItem **each_item_per_category** (unsigned long category_id)

Description: Iterates over all items inside the database which are associated with the given category

category_id: The ID of the category for which all associated items should be retrieved

(2) Signature: *[iter]* RdbItem **each_item_per_category** (unsigned long category_id)

Description: Iterates over all items inside the database which are associated with the given category (non-const version)

category_id: The ID of the category for which all associated items should be retrieved

This non-const variant has been added in version 0.29.

each_item_per_cell

(1) Signature: *[const,iter]* RdbItem **each_item_per_cell** (unsigned long cell_id)

Description: Iterates over all items inside the database which are associated with the given cell

cell_id: The ID of the cell for which all associated items should be retrieved

(2) Signature: *[iter]* RdbItem **each_item_per_cell** (unsigned long cell_id)

Description: Iterates over all items inside the database which are associated with the given cell (non-const version)

cell_id: The ID of the cell for which all associated items should be retrieved

This non-const variant has been added in version 0.29.

each_item_per_cell_and_category

(1) Signature: *[const,iter]* RdbItem **each_item_per_cell_and_category** (unsigned long cell_id, unsigned long category_id)

Description: Iterates over all items inside the database which are associated with the given cell and category

cell_id: The ID of the cell for which all associated items should be retrieved

category_id: The ID of the category for which all associated items should be retrieved

(2) Signature: *[iter]* RdbItem **each_item_per_cell_and_category** (unsigned long cell_id, unsigned long category_id)

Description: Iterates over all items inside the database which are associated with the given cell and category

cell_id: The ID of the cell for which all associated items should be retrieved

category_id: The ID of the category for which all associated items should be retrieved

This non-const variant has been added in version 0.29.

filename

Signature: *[const]* string **filename**

Description: Gets the file name and path where the report database is stored

Returns: The file name and path

This property is set when a database is saved or loaded. It cannot be set manually.

generator

Signature: *[const]* string **generator**

Description: Gets the databases generator

Returns: The generator string

The generator string describes how the database was created, i.e. DRC tool name and tool options. In a later version this will allow re-running the tool that created the report.

Python specific notes:
The object exposes a readable attribute 'generator'. This is the getter.

generator=

Signature: void **generator=** (string generator)

Description: Sets the generator string

generator: The generator string

Python specific notes:
The object exposes a writable attribute 'generator'. This is the setter.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_modified?

Signature: *[const]* bool **is_modified?**

Description: Returns a value indicating whether the database has been modified

load

Signature: void **load** (string filename)

Description: Loads the database from the given file

filename: The file from which to load the database

The reader recognizes the format automatically and will choose the appropriate decoder. 'gzip' compressed files are uncompressed automatically.

name

Signature: *[const]* string **name**

Description: Gets the database name

Returns: The database name

The name of the database is supposed to identify the database within a layout view context. The name is modified to be unique when a database is entered into a layout view.

new

Signature: *[static]* new ReportDatabase ptr **new** (string name)

Description: Creates a report database

name: The name of the database

The name of the database will be used in the user interface to refer to a certain database.

Python specific notes:
This method is the default initializer of the object.

num_items

(1) Signature: *[const]* unsigned long **num_items**

Description: Returns the number of items inside the database

Returns: The total number of items

(2) Signature: *[const]* unsigned long **num_items** (unsigned long cell_id, unsigned long category_id)

Description: Returns the number of items inside the database for a given cell/category combination

cell_id: The ID of the cell for which to retrieve the number

category_id: The ID of the category for which to retrieve the number

Returns: The total number of items for the given cell and the given category

num_items_visited

(1) Signature: *[const]* unsigned long **num_items_visited**

Description: Returns the number of items already visited inside the database

Returns: The total number of items already visited

(2) Signature: *[const]* unsigned long **num_items_visited** (unsigned long cell_id, unsigned long category_id)

Description: Returns the number of items visited already for a given cell/category combination

cell_id: The ID of the cell for which to retrieve the number

category_id: The ID of the category for which to retrieve the number

Returns: The total number of items visited for the given cell and the given category

original_file

Signature: *[const]* string **original_file**

Description: Gets the original file name and path

Returns: The original file name and path

The original file name is supposed to describe the file from which this report database was generated.

Python specific notes:
The object exposes a readable attribute 'original_file'. This is the getter.

original_file=

Signature: void **original_file=** (string path)

Description: Sets the original file name and path

path: The path

Python specific notes:

The object exposes a writable attribute 'original_file'. This is the setter.

reset_modified

Signature: void **reset_modified**

Description: Reset the modified flag

save

Signature: void **save** (string filename)

Description: Saves the database to the given file

filename: The file to which to save the database

The database is always saved in KLayout's XML-based format.

set_item_visited

Signature: void **set_item_visited** (const RdbItem ptr item, bool visited)

Description: Modifies the visited state of an item

item: The item to modify

visited: True to set the item to visited state, false otherwise

set_tag_description

Signature: void **set_tag_description** (unsigned long tag_id, string description)

Description: Sets the tag description for the given tag ID

tag_id: The ID of the tag

description: The description string

See tag_id for a details about tags.

tag_description

Signature: *[const]* string **tag_description** (unsigned long tag_id)

Description: Gets the tag description for the given tag ID

tag_id: The ID of the tag

Returns: The description string

See tag_id for a details about tags.

tag_id

Signature: *[const]* unsigned long **tag_id** (string name)

Description: Gets the tag ID for a given tag name

name: The tag name

Returns: The corresponding tag ID

Tags are used to tag items in the database and to specify tagged (named) values. This method will always succeed and the tag will be created if it does not exist yet. Tags are basically names. There are user tags (for free assignment) and system tags which are used within the system. Both are separated to avoid name clashes.

tag_id handles system tags while user_tag_id handles user tags.

tag_name

Signature: *[const]* string **tag_name** (unsigned long tag_id)

Description: Gets the tag name for the given tag ID

tag_id: The ID of the tag

Returns: The name of the tag

See tag_id for a details about tags.

This method has been introduced in version 0.24.10.

top_cell_name

Signature: *[const]* string **top_cell_name**

Description: Gets the top cell name

Returns: The top cell name

The top cell name identifies the top cell of the design for which the report was generated. This property must be set to establish a proper hierarchical context for a hierarchical report database.

Python specific notes:
The object exposes a readable attribute 'top_cell_name'. This is the getter.

top_cell_name=

Signature: void **top_cell_name=** (string cell_name)

Description: Sets the top cell name string

cell_name: The top cell name

Python specific notes:
The object exposes a writable attribute 'top_cell_name'. This is the setter.

user_tag_id

Signature: *[const]* unsigned long **user_tag_id** (string name)

Description: Gets the tag ID for a given user tag name

name: The user tag name

Returns: The corresponding tag ID

This method will always succeed and the tag will be created if it does not exist yet. See tag_id for a details about tags.

This method has been added in version 0.24.

variants

Signature: unsigned long[] **variants** (string name)

Description: Gets the variants for a given cell name

name: The basic name of the cell

Returns: An array of ID's representing cells that are variants for the given base name

4.195. API reference - Class LayerProperties

Notation used in Ruby API documentation

Module: lay

Description: The layer properties structure

The layer properties encapsulate the settings relevant for the display and source of a layer.

Each attribute is present in two incarnations: local and real. "real" refers to the effective attribute after collecting the attributes from the parents to the leaf property node. In the spirit of this distinction, all read accessors are present in "local" and "real" form. The read accessors take a boolean parameter "real" that must be set to true, if the real value shall be returned.

"brightness" is a index that indicates how much to make the color brighter to darker rendering the effective color (eff_frame_color, eff_fill_color). It's value is roughly between -255 and 255.

Public constructors

new LayerProperties ptr	**new**		Creates a new object of this class

Public methods

[const]	bool	**!=**	(const LayerProperties other)	Inequality
[const]	bool	**==**	(const LayerProperties other)	Equality
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	int	**animation**	(bool real)	Gets the animation state
[const]	int	**animation**		Gets the animation state
	void	**animation=**	(int animation)	Sets the animation state
	void	**assign**	(const LayerProperties other)	Assigns another object to self
[const]	int	**cellview**		Gets the the cellview index

	void	clear_dither_pattern		Clears the dither pattern
	void	clear_fill_color		Resets the fill color
	void	clear_frame_color		Resets the frame color
	void	clear_line_style		Clears the line style
	void	clear_lower_hier_level		Clears the lower hierarchy level specification
	void	clear_source_name		Removes any stream layer name specification from this layer
	void	clear_upper_hier_level		Clears the upper hierarchy level specification
[const]	int	dither_pattern	(bool real)	Gets the dither pattern index
[const]	int	dither_pattern		Gets the dither pattern index
	void	dither_pattern=	(int index)	Sets the dither pattern index
[const]	new LayerProperties ptr	dup		Creates a copy of self
[const]	unsigned int	eff_dither_pattern	(bool real)	Gets the effective dither pattern index
[const]	unsigned int	eff_dither_pattern		Gets the effective dither pattern index
[const]	unsigned int	eff_fill_color	(bool real)	Gets the effective fill color
[const]	unsigned int	eff_fill_color		Gets the effective fill color
[const]	unsigned int	eff_frame_color	(bool real)	Gets the effective frame color
[const]	unsigned int	eff_frame_color		Gets the effective frame color
[const]	unsigned int	eff_line_style	(bool real)	Gets the effective line style index
[const]	unsigned int	eff_line_style		Gets the line style index
[const]	int	fill_brightness	(bool real)	Gets the fill brightness value
[const]	int	fill_brightness		Gets the fill brightness value
	void	fill_brightness=	(int brightness)	Sets the fill brightness
[const]	unsigned int	fill_color	(bool real)	Gets the fill color
[const]	unsigned int	fill_color		Gets the fill color
	void	fill_color=	(unsigned int color)	Sets the fill color to the given value
[const]	LayerProperties	flat		Returns the "flattened" (effective) layer properties entry for this node

[const]	int	**frame_brightness**	(bool real)	Gets the frame brightness value
[const]	int	**frame_brightness**		Gets the frame brightness value
	void	**frame_brightness=**	(int brightness)	Sets the frame brightness
[const]	unsigned int	**frame_color**	(bool real)	Gets the frame color
[const]	unsigned int	**frame_color**		Gets the frame color
	void	**frame_color=**	(unsigned int color)	Sets the frame color to the given value
[const]	bool	**has_dither_pattern?**	(bool real)	True, if the dither pattern is set
[const]	bool	**has_dither_pattern?**		True, if the dither pattern is set
[const]	bool	**has_fill_color?**	(bool real)	True, if the fill color is set
[const]	bool	**has_fill_color?**		True, if the fill color is set
[const]	bool	**has_frame_color?**	(bool real)	True, if the frame color is set
[const]	bool	**has_frame_color?**		True, if the frame color is set
[const]	bool	**has_line_style?**	(bool real)	Gets a value indicating whether the line style is set
[const]	bool	**has_line_style?**		True, if the line style is set
[const]	bool	**has_lower_hier_level?**	(bool real)	Gets a value indicating whether a lower hierarchy level is explicitly specified
[const]	bool	**has_lower_hier_level?**		Gets a value indicating whether a lower hierarchy level is explicitly specified
[const]	bool	**has_source_name?**	(bool real)	Gets a value indicating whether a stream layer name is specified for this layer
[const]	bool	**has_source_name?**		Gets a value indicating whether a stream layer name is specified for this layer
[const]	bool	**has_upper_hier_level?**	(bool real)	Gets a value indicating whether an upper hierarchy level is explicitly specified
[const]	bool	**has_upper_hier_level?**		Gets a value indicating whether an upper hierarchy level is explicitly specified
[const]	int	**layer_index**		Gets the the layer index
[const]	int	**line_style**	(bool real)	Gets the line style index
[const]	int	**line_style**		Gets the line style index
	void	**line_style=**	(int index)	Sets the line style index
[const]	int	**lower_hier_level**	(bool real)	Gets the lower hierarchy level shown

[const]	int	**lower_hier_level**		Gets the lower hierarchy level shown
	void	**lower_hier_level=**	(int level)	Sets the lower hierarchy level
[const]	int	**lower_hier_level_mode**	(bool real)	Gets the mode for the lower hierarchy level.
[const]	int	**lower_hier_level_mode**		Gets the mode for the lower hierarchy level.
[const]	bool	**lower_hier_level_relative?**	(bool real)	Gets a value indicating whether the lower hierarchy level is relative.
[const]	bool	**lower_hier_level_relative?**		Gets a value indicating whether the upper hierarchy level is relative.
	void	**marked=**	(bool marked)	Sets the marked state
[const]	bool	**marked?**	(bool real)	Gets the marked state
[const]	bool	**marked?**		Gets the marked state
[const]	string	**name**		Gets the name
	void	**name=**	(string name)	Sets the name to the given string
	void	**set_lower_hier_level**	(int level, bool relative)	Sets the lower hierarchy level and if it is relative to the context cell
	void	**set_lower_hier_level**	(int level, bool relative, int mode)	Sets the lower hierarchy level, whether it is relative to the context cell and the mode
	void	**set_upper_hier_level**	(int level, bool relative)	Sets the upper hierarchy level and if it is relative to the context cell
	void	**set_upper_hier_level**	(int level, bool relative, int mode)	Sets the upper hierarchy level, if it is relative to the context cell and the mode
[const]	string	**source**	(bool real)	Gets the source specification
[const]	string	**source**		Gets the source specification
	void	**source=**	(string s)	Loads the source specification from a string
[const]	int	**source_cellview**	(bool real)	Gets the cellview index that this layer refers to
[const]	int	**source_cellview**		Gets the cellview index that this layer refers to
	void	**source_cellview=**	(int cellview_index)	Sets the cellview index that this layer refers to
[const]	int	**source_datatype**	(bool real)	Gets the stream datatype that the shapes are taken from
[const]	int	**source_datatype**		Gets the stream datatype that the shapes are taken from

	void	source_datatype=	(int datatype)	Sets the stream datatype that the shapes are taken from
[const]	int	source_layer	(bool real)	Gets the stream layer that the shapes are taken from
[const]	int	source_layer		Gets the stream layer that the shapes are taken from
	void	source_layer=	(int layer)	Sets the stream layer that the shapes are taken from
[const]	int	source_layer_index	(bool real)	Gets the layer index that the shapes are taken from
[const]	int	source_layer_index		Gets the stream layer that the shapes are taken from
	void	source_layer_index=	(int index)	Sets the layer index specification that the shapes are taken from
[const]	string	source_name	(bool real)	Gets the stream name that the shapes are taken from
[const]	string	source_name		Gets the stream name that the shapes are taken from
	void	source_name=	(string name)	Sets the stream layer name that the shapes are taken from
[const]	DCplxTrans[]	trans	(bool real)	Gets the transformations that the layer is transformed with
[const]	DCplxTrans[]	trans		Gets the transformations that the layer is transformed with
	void	trans=	(DCplxTrans[] trans_vector)	Sets the transformations that the layer is transformed with
	void	transparent=	(bool transparent)	Sets the transparency state
[const]	bool	transparent?	(bool real)	Gets the transparency state
[const]	bool	transparent?		Gets the transparency state
[const]	int	upper_hier_level	(bool real)	Gets the upper hierarchy level shown
[const]	int	upper_hier_level		Gets the upper hierarchy level shown
	void	upper_hier_level=	(int level)	Sets a upper hierarchy level
[const]	int	upper_hier_level_mode	(bool real)	Gets the mode for the upper hierarchy level.
[const]	int	upper_hier_level_mode		Gets the mode for the upper hierarchy level.

[const]	bool	**upper_hier_level_relative?**	(bool real)	Gets a value indicating whether if the upper hierarchy level is relative.
[const]	bool	**upper_hier_level_relative?**		Gets a value indicating whether the upper hierarchy level is relative.
	void	**valid=**	(bool valid)	Sets the validity state
[const]	bool	**valid?**	(bool real)	Gets the validity state
[const]	bool	**valid?**		Gets the validity state
	void	**visible=**	(bool visible)	Sets the visibility state
[const]	bool	**visible?**	(bool real)	Gets the visibility state
[const]	bool	**visible?**		Gets the visibility state
[const]	int	**width**	(bool real)	Gets the line width
[const]	int	**width**		Gets the line width
	void	**width=**	(int width)	Sets the line width to the given width
	void	**xfill=**	(bool xfill)	Sets a value indicating whether shapes are drawn with a cross
[const]	bool	**xfill?**	(bool real)	Gets a value indicating whether shapes are drawn with a cross
[const]	bool	**xfill?**		Gets a value indicating whether shapes are drawn with a cross

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	bool	**lower_hier_level_relative**	(bool real)	Use of this method is deprecated. Use lower_hier_level_relative? instead
[const]	bool	**lower_hier_level_relative**		Use of this method is deprecated. Use lower_hier_level_relative? instead
[const]	bool	**upper_hier_level_relative**	(bool real)	Use of this method is deprecated. Use upper_hier_level_relative? instead

[const]	bool	**upper_hier_level_relative**	Use of this method is deprecated. Use upper_hier_level_relative? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const LayerProperties other)

Description: Inequality

other: The other object to compare against

==

Signature: *[const]* bool **==** (const LayerProperties other)

Description: Equality

other: The other object to compare against

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

animation

(1) Signature: *[const]* int **animation** (bool real)

Description: Gets the animation state

The animation state is an integer either being 0 (static), 1 (scrolling), 2 (blinking) or 3 (inversely blinking)

Python specific notes:
This method is available as 'animation_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **animation**

Description: Gets the animation state

This method is a convenience method for "animation(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'animation'. This is the getter.

animation=

Signature: void **animation=** (int animation)

Description: Sets the animation state

See the description of the animation method for details about the animation state

Python specific notes:
The object exposes a writable attribute 'animation'. This is the setter.

assign

Signature: void **assign** (const LayerProperties other)

Description: Assigns another object to self

cellview

Signature: *[const]* int **cellview**

Description: Gets the the cellview index

This is the index of the actual cellview to use. Basically, this method returns source_cellview in "real" mode. The result may be different, if the cellview is not valid for example. In this case, a negative value is returned.

clear_dither_pattern

Signature: void **clear_dither_pattern**

Description: Clears the dither pattern

clear_fill_color

Signature: void **clear_fill_color**

Description: Resets the fill color

clear_frame_color

Signature: void **clear_frame_color**

Description: Resets the frame color

clear_line_style

Signature: void **clear_line_style**

Description: Clears the line style

This method has been introduced in version 0.25.

clear_lower_hier_level

Signature: void **clear_lower_hier_level**

Description: Clears the lower hierarchy level specification

See has_lower_hier_level for a description of this property

clear_source_name

Signature: void **clear_source_name**

Description: Removes any stream layer name specification from this layer

clear_upper_hier_level

Signature: void **clear_upper_hier_level**

Description: Clears the upper hierarchy level specification

See has_upper_hier_level for a description of this property

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dither_pattern

(1) Signature: *[const]* int **dither_pattern** (bool real)

Description: Gets the dither pattern index

real: Set to true to return the real instead of local value

This method may deliver an invalid dither pattern index if it is not set.

Python specific notes:
This method is available as 'dither_pattern_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **dither_pattern**

Description: Gets the dither pattern index

This method is a convenience method for "dither_pattern(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'dither_pattern'. This is the getter.

dither_pattern=

Signature: void **dither_pattern=** (int index)

Description: Sets the dither pattern index

The dither pattern index must be one of the valid indices. The first indices are reserved for built-in pattern, the following ones are custom pattern. Index 0 is always solid filled and 1 is always the hollow filled pattern. For custom pattern see LayoutView#add_stipple.

Python specific notes:
The object exposes a writable attribute 'dither_pattern'. This is the setter.

dup

Signature: *[const]* new LayerProperties ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

eff_dither_pattern

(1) Signature: *[const]* unsigned int **eff_dither_pattern** (bool real)

Description: Gets the effective dither pattern index

real: Set to true to return the real instead of local value

The effective dither pattern index is always a valid index, even if no dither pattern is set.

(2) Signature: *[const]* unsigned int **eff_dither_pattern**

Description: Gets the effective dither pattern index

This method is a convenience method for "eff_dither_pattern(true)"

This method has been introduced in version 0.22.

eff_fill_color

(1) Signature: *[const]* unsigned int **eff_fill_color** (bool real)

Description: Gets the effective fill color

real: Set to true to return the real instead of local value

The effective fill color is computed from the frame color brightness and the frame color.

(2) Signature: *[const]* unsigned int **eff_fill_color**

Description: Gets the effective fill color

This method is a convenience method for "eff_fill_color(true)"

This method has been introduced in version 0.22.

eff_frame_color

(1) Signature: *[const]* unsigned int **eff_frame_color** (bool real)

Description: Gets the effective frame color

real: Set to true to return the real instead of local value

The effective frame color is computed from the frame color brightness and the frame color.

(2) Signature: *[const]* unsigned int **eff_frame_color**

Description: Gets the effective frame color

This method is a convenience method for "eff_frame_color(true)"

This method has been introduced in version 0.22.

eff_line_style

(1) Signature: *[const]* unsigned int **eff_line_style** (bool real)

Description: Gets the effective line style index

real: Set to true to return the real instead of local value

The effective line style index is always a valid index, even if no line style is set. In that case, a default style index will be returned.

This method has been introduced in version 0.25.

(2) Signature: *[const]* unsigned int **eff_line_style**

Description: Gets the line style index

This method is a convenience method for "eff_line_style(true)"

This method has been introduced in version 0.25.

fill_brightness

(1) Signature: *[const]* int **fill_brightness** (bool real)

Description: Gets the fill brightness value

real: Set to true to return the real instead of local value

If the brightness is not set, this method may return an invalid value

Python specific notes:
This method is available as 'fill_brightness_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **fill_brightness**

Description: Gets the fill brightness value

This method is a convenience method for "fill_brightness(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'fill_brightness'. This is the getter.

fill_brightness=

Signature: void **fill_brightness=** (int brightness)

Description: Sets the fill brightness

For neutral brightness set this value to 0. For darker colors set it to a negative value (down to -255), for brighter colors to a positive value (up to 255)

Python specific notes:
The object exposes a writable attribute 'fill_brightness'. This is the setter.

fill_color

(1) Signature: *[const]* unsigned int **fill_color** (bool real)

Description: Gets the fill color

real: Set to true to return the real instead of local value

This method may return an invalid color if the color is not set.

Python specific notes:
This method is available as 'fill_color_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* unsigned int **fill_color**

Description: Gets the fill color

This method is a convenience method for "fill_color(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'fill_color'. This is the getter.

fill_color=

Signature: void **fill_color=** (unsigned int color)

Description: Sets the fill color to the given value

The color is a 32bit value encoding the blue value in the lower 8 bits, the green value in the next 8 bits and the red value in the 8 bits above that.

Python specific notes:
The object exposes a writable attribute 'fill_color'. This is the setter.

flat

Signature: *[const]* LayerProperties **flat**

Description: Returns the "flattened" (effective) layer properties entry for this node

This method returns a LayerProperties object that is not embedded into a hierarchy. This object represents the effective layer properties for the given node. In particular, all 'local' properties are identical to the 'real' properties. Such an object can be used as a basis for manipulations. This method has been introduced in version 0.22.

frame_brightness

(1) Signature: *[const]* int **frame_brightness** (bool real)

Description: Gets the frame brightness value

real: Set to true to return the real instead of local value

If the brightness is not set, this method may return an invalid value

Python specific notes:
This method is available as 'frame_brightness_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **frame_brightness**

Description: Gets the frame brightness value

This method is a convenience method for "frame_brightness(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'frame_brightness'. This is the getter.

frame_brightness=

Signature: void **frame_brightness=** (int brightness)

Description: Sets the frame brightness

For neutral brightness set this value to 0. For darker colors set it to a negative value (down to -255), for brighter colors to a positive value (up to 255)

Python specific notes:
The object exposes a writable attribute 'frame_brightness'. This is the setter.

frame_color

(1) Signature: *[const]* unsigned int **frame_color** (bool real)

Description: Gets the frame color

real: Set to true to return the real instead of local value

This method may return an invalid color if the color is not set.

Python specific notes:
This method is available as 'frame_color_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* unsigned int **frame_color**

Description: Gets the frame color

This method is a convenience method for "frame_color(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'frame_color'. This is the getter.

frame_color=

Signature: void **frame_color=** (unsigned int color)

Description: Sets the frame color to the given value

The color is a 32bit value encoding the blue value in the lower 8 bits, the green value in the next 8 bits and the red value in the 8 bits above that.

Python specific notes:
The object exposes a writable attribute 'frame_color'. This is the setter.

has_dither_pattern?

(1) Signature: *[const]* bool **has_dither_pattern?** (bool real)

Description: True, if the dither pattern is set

(2) Signature: *[const]* bool **has_dither_pattern?**

Description: True, if the dither pattern is set

This method is a convenience method for "has_dither_pattern?(true)"

This method has been introduced in version 0.22.

has_fill_color?

(1) Signature: *[const]* bool **has_fill_color?** (bool real)

Description: True, if the fill color is set

(2) Signature: *[const]* bool **has_fill_color?**

Description: True, if the fill color is set

This method is a convenience method for "has_fill_color?(true)"

This method has been introduced in version 0.22.

has_frame_color?

(1) Signature: *[const]* bool **has_frame_color?** (bool real)

Description: True, if the frame color is set

(2) Signature: *[const]* bool **has_frame_color?**
Description: True, if the frame color is set

This method is a convenience method for "has_frame_color?(true)"
This method has been introduced in version 0.22.

has_line_style?

(1) Signature: *[const]* bool **has_line_style?** (bool real)
Description: Gets a value indicating whether the line style is set

This method has been introduced in version 0.25.

(2) Signature: *[const]* bool **has_line_style?**
Description: True, if the line style is set

This method is a convenience method for "has_line_style?(true)"
This method has been introduced in version 0.25.

has_lower_hier_level?

(1) Signature: *[const]* bool **has_lower_hier_level?** (bool real)
Description: Gets a value indicating whether a lower hierarchy level is explicitly specified

If "real" is true, the effective value is returned.

(2) Signature: *[const]* bool **has_lower_hier_level?**
Description: Gets a value indicating whether a lower hierarchy level is explicitly specified

This method is a convenience method for "has_lower_hier_level?(true)"
This method has been introduced in version 0.22.

has_source_name?

(1) Signature: *[const]* bool **has_source_name?** (bool real)
Description: Gets a value indicating whether a stream layer name is specified for this layer

If "real" is true, the effective value is returned.

(2) Signature: *[const]* bool **has_source_name?**
Description: Gets a value indicating whether a stream layer name is specified for this layer

This method is a convenience method for "has_source_name?(true)"
This method has been introduced in version 0.22.

has_upper_hier_level?

(1) Signature: *[const]* bool **has_upper_hier_level?** (bool real)
Description: Gets a value indicating whether an upper hierarchy level is explicitly specified

If "real" is true, the effective value is returned.

(2) Signature: *[const]* bool **has_upper_hier_level?**
Description: Gets a value indicating whether an upper hierarchy level is explicitly specified

This method is a convenience method for "has_upper_hier_level?(true)"
This method has been introduced in version 0.22.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

layer_index

Signature: *[const]* int **layer_index**

Description: Gets the the layer index

This is the index of the actual layer used. The source specification given by source_layer, source_datatype, source_name is evaluated and the corresponding layer is looked up in the layout object. If a source_layer_index is specified, this layer index is taken as the layer index to use.

line_style

(1) Signature: *[const]* int **line_style** (bool real)

Description: Gets the line style index

real: Set to true to return the real instead of local value

This method may deliver an invalid line style index if it is not set (see has_line_style?).

This method has been introduced in version 0.25.

Python specific notes:
This method is available as 'line_style_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **line_style**

Description: Gets the line style index

This method is a convenience method for "line_style(true)"

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'line_style'. This is the getter.

line_style=

Signature: void **line_style=** (int index)

Description: Sets the line style index

The line style index must be one of the valid indices. The first indices are reserved for built-in pattern, the following ones are custom pattern. Index 0 is always solid filled. For custom line styles see LayoutView#add_line_style.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'line_style'. This is the setter.

lower_hier_level

(1) Signature: *[const]* int **lower_hier_level** (bool real)

Description: Gets the lower hierarchy level shown

This is the hierarchy level at which the drawing starts. This property is only meaningful, if has_lower_hier_level is true. The hierarchy level can be relative in which case, 0 refers to the context cell's level. A mode can be specified for the hierarchy level which is 0 for absolute, 1 for minimum of specified level and set level and 2 for maximum of specified level and set level.

Python specific notes:

This method is available as 'lower_hier_level_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **lower_hier_level**

Description: Gets the lower hierarchy level shown

This method is a convenience method for "lower_hier_level(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'lower_hier_level'. This is the getter.

lower_hier_level=

Signature: void **lower_hier_level=** (int level)

Description: Sets the lower hierarchy level

If this method is called, the lower hierarchy level is enabled. See lower_hier_level for a description of this property.

Python specific notes:
The object exposes a writable attribute 'lower_hier_level'. This is the setter.

lower_hier_level_mode

(1) Signature: *[const]* int **lower_hier_level_mode** (bool real)

Description: Gets the mode for the lower hierarchy level.

real:	If true, the computed value is returned, otherwise the local node value

The mode value can be 0 (value is given by lower_hier_level), 1 for "minimum value" and 2 for "maximum value".

This method has been introduced in version 0.20.

(2) Signature: *[const]* int **lower_hier_level_mode**

Description: Gets the mode for the lower hierarchy level.

This method is a convenience method for "lower_hier_level_mode(true)"

This method has been introduced in version 0.22.

lower_hier_level_relative

(1) Signature: *[const]* bool **lower_hier_level_relative** (bool real)

Description: Gets a value indicating whether the lower hierarchy level is relative.

Use of this method is deprecated. Use lower_hier_level_relative? instead

See lower_hier_level for a description of this property.

This method has been introduced in version 0.19.

(2) Signature: *[const]* bool **lower_hier_level_relative**

Description: Gets a value indicating whether the upper hierarchy level is relative.

Use of this method is deprecated. Use lower_hier_level_relative? instead

This method is a convenience method for "lower_hier_level_relative(true)"

This method has been introduced in version 0.22.

lower_hier_level_relative?

(1) Signature: *[const]* bool **lower_hier_level_relative?** (bool real)

Description: Gets a value indicating whether the lower hierarchy level is relative.

See lower_hier_level for a description of this property.

This method has been introduced in version 0.19.

(2) Signature: *[const]* bool **lower_hier_level_relative?**

Description: Gets a value indicating whether the upper hierarchy level is relative.

This method is a convenience method for "lower_hier_level_relative(true)"

This method has been introduced in version 0.22.

marked=

Signature: void **marked=** (bool marked)

Description: Sets the marked state

Python specific notes:
The object exposes a writable attribute 'marked'. This is the setter.

marked?

(1) Signature: *[const]* bool **marked?** (bool real)

Description: Gets the marked state

Python specific notes:
This method is available as 'marked_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* bool **marked?**

Description: Gets the marked state

This method is a convenience method for "marked?(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'marked'. This is the getter.

name

Signature: *[const]* string **name**

Description: Gets the name

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string name)

Description: Sets the name to the given string

Python specific notes:
The object exposes a writable attribute 'name'. This is the setter.

new

Signature: *[static]* new LayerProperties ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

set_lower_hier_level

(1) Signature: void **set_lower_hier_level** (int level, bool relative)

Description: Sets the lower hierarchy level and if it is relative to the context cell

If this method is called, the lower hierarchy level is enabled. See lower_hier_level for a description of this property.

This method has been introduced in version 0.19.

(2) Signature: void **set_lower_hier_level** (int level, bool relative, int mode)

Description: Sets the lower hierarchy level, whether it is relative to the context cell and the mode

If this method is called, the lower hierarchy level is enabled. See lower_hier_level for a description of this property.

This method has been introduced in version 0.20.

set_upper_hier_level

(1) Signature: void **set_upper_hier_level** (int level, bool relative)

Description: Sets the upper hierarchy level and if it is relative to the context cell

If this method is called, the upper hierarchy level is enabled. See upper_hier_level for a description of this property.

This method has been introduced in version 0.19.

(2) Signature: void **set_upper_hier_level** (int level, bool relative, int mode)

Description: Sets the upper hierarchy level, if it is relative to the context cell and the mode

If this method is called, the upper hierarchy level is enabled. See upper_hier_level for a description of this property.

This method has been introduced in version 0.20.

source

(1) Signature: *[const]* string **source** (bool real)

Description: Gets the source specification

real: Set to true to return the computed instead of local value

This method delivers the source specification as a string

Python specific notes:
This method is available as 'source_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* string **source**

Description: Gets the source specification

This method is a convenience method for "source(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'source'. This is the getter.

source=

Signature: void **source=** (string s)

Description: Loads the source specification from a string

Sets the source specification to the given string. The source specification may contain the cellview index, the source layer (given by layer/datatype or layer name), transformation, property selector etc. This method throws an exception if the specification is not valid.

Python specific notes:
The object exposes a writable attribute 'source'. This is the setter.

source_cellview

(1) Signature: *[const]* int **source_cellview** (bool real)

Description: Gets the cellview index that this layer refers to

If "real" is true, the effective value is returned.

Python specific notes:
This method is available as 'source_cellview_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **source_cellview**

Description: Gets the cellview index that this layer refers to

This method is a convenience method for "source_cellview(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'source_cellview'. This is the getter.

source_cellview=

Signature: void **source_cellview=** (int cellview_index)

Description: Sets the cellview index that this layer refers to

See cellview for a description of the transformations.

Python specific notes:
The object exposes a writable attribute 'source_cellview'. This is the setter.

source_datatype

(1) Signature: *[const]* int **source_datatype** (bool real)

Description: Gets the stream datatype that the shapes are taken from

If the datatype is positive, the actual layer is looked up by this stream datatype. If a name or layer index is specified, the stream datatype is not used.

If "real" is true, the effective value is returned.

Python specific notes:
This method is available as 'source_datatype_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **source_datatype**

Description: Gets the stream datatype that the shapes are taken from

This method is a convenience method for "source_datatype(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'source_datatype'. This is the getter.

source_datatype=

Signature: void **source_datatype=** (int datatype)

Description: Sets the stream datatype that the shapes are taken from

See datatype for a description of this property

Python specific notes:
The object exposes a writable attribute 'source_datatype'. This is the setter.

source_layer

(1) Signature: *[const]* int **source_layer** (bool real)

Description: Gets the stream layer that the shapes are taken from

If the layer is positive, the actual layer is looked up by this stream layer. If a name or layer index is specified, the stream layer is not used.

If "real" is true, the effective value is returned.

Python specific notes:
This method is available as 'source_layer_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **source_layer**

Description: Gets the stream layer that the shapes are taken from

This method is a convenience method for "source_layer(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'source_layer'. This is the getter.

source_layer=

Signature: void **source_layer=** (int layer)

Description: Sets the stream layer that the shapes are taken from

See source_layer for a description of this property

Python specific notes:
The object exposes a writable attribute 'source_layer'. This is the setter.

source_layer_index

(1) Signature: *[const]* int **source_layer_index** (bool real)

Description: Gets the layer index that the shapes are taken from

If the layer index is positive, the shapes drawn are taken from this layer rather than searched for by layer and datatype. This property is stronger than the layer/datatype or name specification.

A different method is layer_index which indicates the ID of the layer actually used. While "source_layer_index" is one of several ways to address the layer drawn, "layer_index" is the ID (index) of the layer matching the source specification and is >= 0 if such a layer is found.

If "real" is true, the effective value is returned.

Python specific notes:
This method is available as 'source_layer_index_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **source_layer_index**

Description: Gets the stream layer that the shapes are taken from

This method is a convenience method for "source_layer_index(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'source_layer_index'. This is the getter.

source_layer_index=

Signature: void **source_layer_index=** (int index)

Description: Sets the layer index specification that the shapes are taken from

See source_layer_index for a description of this property.

Python specific notes:
The object exposes a writable attribute 'source_layer_index'. This is the setter.

source_name

(1) Signature: *[const]* string **source_name** (bool real)

Description: Gets the stream name that the shapes are taken from

If the name is non-empty, the actual layer is looked up by this stream layer name. If a layer index (see layer_index) is specified, the stream datatype is not used. A name is only meaningful for OASIS files.

If "real" is true, the effective value is returned.

Python specific notes:
This method is available as 'source_name_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* string **source_name**

Description: Gets the stream name that the shapes are taken from

This method is a convenience method for "source_name(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'source_name'. This is the getter.

source_name=

Signature: void **source_name=** (string name)

Description: Sets the stream layer name that the shapes are taken from

See name for a description of this property

Python specific notes:
The object exposes a writable attribute 'source_name'. This is the setter.

trans

(1) Signature: *[const]* DCplxTrans[] **trans** (bool real)

Description: Gets the transformations that the layer is transformed with

The transformations returned by this accessor is the one used for displaying this layer. The layout is transformed with each of these transformations before it is drawn.

If "real" is true, the effective value is returned.

Python specific notes:
This method is available as 'trans_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* DCplxTrans[] **trans**

Description: Gets the transformations that the layer is transformed with

This method is a convenience method for "trans(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'trans'. This is the getter.

trans=

Signature: void **trans=** (DCplxTrans[] trans_vector)

Description: Sets the transformations that the layer is transformed with

See trans for a description of the transformations.

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.

transparent=

Signature: void **transparent=** (bool transparent)

Description: Sets the transparency state

Python specific notes:
The object exposes a writable attribute 'transparent'. This is the setter.

transparent?

(1) Signature: *[const]* bool **transparent?** (bool real)

Description: Gets the transparency state

Python specific notes:
This method is available as 'transparent_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* bool **transparent?**

Description: Gets the transparency state

This method is a convenience method for "transparent?(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'transparent'. This is the getter.

upper_hier_level

(1) Signature: *[const]* int **upper_hier_level** (bool real)

Description: Gets the upper hierarchy level shown

This is the hierarchy level at which the drawing starts. This property is only meaningful, if has_upper_hier_level is true. The hierarchy level can be relative in which case, 0 refers to the context cell's level. A mode can be specified for the hierarchy level which is 0 for absolute, 1 for minimum of specified level and set level and 2 for maximum of specified level and set level.

Python specific notes:
This method is available as 'upper_hier_level_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **upper_hier_level**

Description: Gets the upper hierarchy level shown

This method is a convenience method for "upper_hier_level(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'upper_hier_level'. This is the getter.

upper_hier_level=

Signature: void **upper_hier_level=** (int level)

Description: Sets a upper hierarchy level

If this method is called, the upper hierarchy level is enabled. See upper_hier_level for a description of this property.

Python specific notes:
The object exposes a writable attribute 'upper_hier_level'. This is the setter.

upper_hier_level_mode

(1) Signature: *[const]* int **upper_hier_level_mode** (bool real)

Description: Gets the mode for the upper hierarchy level.

real:	If true, the computed value is returned, otherwise the local node value

The mode value can be 0 (value is given by upper_hier_level), 1 for "minimum value" and 2 for "maximum value".

This method has been introduced in version 0.20.

(2) Signature: *[const]* int **upper_hier_level_mode**

Description: Gets the mode for the upper hierarchy level.

This method is a convenience method for "upper_hier_level_mode(true)"

This method has been introduced in version 0.22.

upper_hier_level_relative

(1) Signature: *[const]* bool **upper_hier_level_relative** (bool real)

Description: Gets a value indicating whether if the upper hierarchy level is relative.

Use of this method is deprecated. Use upper_hier_level_relative? instead

See upper_hier_level for a description of this property.

This method has been introduced in version 0.19.

(2) Signature: *[const]* bool **upper_hier_level_relative**

Description: Gets a value indicating whether the upper hierarchy level is relative.

Use of this method is deprecated. Use upper_hier_level_relative? instead

This method is a convenience method for "upper_hier_level_relative(true)"

This method has been introduced in version 0.22.

upper_hier_level_relative?

(1) Signature: *[const]* bool **upper_hier_level_relative?** (bool real)

Description: Gets a value indicating whether if the upper hierarchy level is relative.

See upper_hier_level for a description of this property.

This method has been introduced in version 0.19.

(2) Signature: *[const]* bool **upper_hier_level_relative?**

Description: Gets a value indicating whether the upper hierarchy level is relative.

This method is a convenience method for "upper_hier_level_relative(true)"

This method has been introduced in version 0.22.

valid=

Signature: void **valid=** (bool valid)

Description: Sets the validity state

Python specific notes:
The object exposes a writable attribute 'valid'. This is the setter.

valid?

(1) Signature: *[const]* bool **valid?** (bool real)

Description: Gets the validity state

Python specific notes:
This method is available as 'valid_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* bool **valid?**

Description: Gets the validity state

This method is a convenience method for "valid?(true)"

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a readable attribute 'valid'. This is the getter.

visible=

Signature: void **visible=** (bool visible)

Description: Sets the visibility state

Python specific notes:
The object exposes a writable attribute 'visible'. This is the setter.

visible?

(1) Signature: *[const]* bool **visible?** (bool real)

Description: Gets the visibility state

Python specific notes:
This method is available as 'visible_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* bool **visible?**

Description: Gets the visibility state

This method is a convenience method for "visible?(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'visible'. This is the getter.

width

(1) Signature: *[const]* int **width** (bool real)

Description: Gets the line width

Python specific notes:
This method is available as 'width_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* int **width**

Description: Gets the line width

This method is a convenience method for "width(true)"

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'width'. This is the getter.

width=

Signature: void **width=** (int width)

Description: Sets the line width to the given width

Python specific notes:
The object exposes a writable attribute 'width'. This is the setter.

xfill=

Signature: void **xfill=** (bool xfill)

Description: Sets a value indicating whether shapes are drawn with a cross

This attribute has been introduced in version 0.25.

Python specific notes:

The object exposes a writable attribute 'xfill'. This is the setter.

xfill?

(1) Signature: *[const]* bool **xfill?** (bool real)

Description: Gets a value indicating whether shapes are drawn with a cross

This attribute has been introduced in version 0.25.

Python specific notes:
This method is available as 'xfill_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* bool **xfill?**

Description: Gets a value indicating whether shapes are drawn with a cross

This method is a convenience method for "xfill?(true)"

This attribute has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'xfill'. This is the getter.

4.196. API reference - Class LayerPropertiesNode

Notation used in Ruby API documentation

Module: lay

Description: A layer properties node structure

Class hierarchy: LayerPropertiesNode » LayerProperties

This class is derived from LayerProperties. Objects of this class are used in the hierarchy of layer views that are arranged in a tree while the LayerProperties object reflects the properties of a single node.

Public methods

[const]	bool	**!=**	(const LayerPropertie other)	Inequality
[const]	bool	**==**	(const LayerPropertiesNode other)	Equality
	void	**_assign**	(const LayerPropertie other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new LayerPropertiesNode ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	LayerPropertiesNodeRef	**add_child**		Add a child entry
	LayerPropertiesNodeRef	**add_child**	(const LayerProperties ptr child)	Add a child entry
[const]	DBox	**bbox**		Compute the bbox of this layer
	void	**clear_children**		Clears all children
	void	**expanded=**	(bool ex)	Set a value indicating whether the layer tree node is expanded.

[const]	LayerPropertiesNode	**flat**	return the "flattened" (effective) layer properties node for this node
[const]	bool	**has_children?**	Test, if there are children
[const]	unsigned int	**id**	Obtain the unique ID
[const]	bool	**is_expanded?**	Gets a value indicating whether the layer tree node is expanded.
[const]	unsigned int	**list_index**	Gets the index of the layer properties list that the node lives in
	LayoutView ptr	**view**	Gets the view this node lives in

Detailed description

!=

Signature: *[const]* bool **!=** (const LayerPropertiesNode other)

Description: Inequality

other: The other object to compare against

==

Signature: *[const]* bool **==** (const LayerPropertiesNode other)

Description: Equality

other: The other object to compare against

_assign

Signature: void **_assign** (const LayerPropertiesNode other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new LayerPropertiesNode ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_child

(1) Signature: LayerPropertiesNodeRef **add_child**

Description: Add a child entry

Returns: A reference to the node created

This method allows building a layer properties tree by adding children to node objects. It returns a reference to the node object created which is a freshly initialized one.

The parameterless version of this method was introduced in version 0.25.

(2) Signature: LayerPropertiesNodeRef **add_child** (const LayerProperties ptr child)

Description: Add a child entry

Returns: A reference to the node created

This method allows building a layer properties tree by adding children to node objects. It returns a reference to the node object created.

This method was introduced in version 0.22.

bbox

Signature: *[const]* DBox **bbox**

Description: Compute the bbox of this layer

Returns: A bbox in micron units

This takes the layout and path definition (supported by the given default layout or path, if no specific is given). The node must have been attached to a view to make this operation possible.

clear_children

Signature: void **clear_children**

Description: Clears all children

This method was introduced in version 0.22.

expanded=

Signature: void **expanded=** (bool ex)

Description: Set a value indicating whether the layer tree node is expanded.

Setting this value to 'true' will expand (open) the tree node. Setting it to 'false' will collapse the node.

This predicate has been introduced in version 0.28.6.

Python specific notes:
The object exposes a writable attribute 'expanded'. This is the setter.

flat

Signature: *[const]* LayerPropertiesNode **flat**

Description: return the "flattened" (effective) layer properties node for this node

This method returns a LayerPropertiesNode object that is not embedded into a hierarchy. This object represents the effective layer properties for the given node. In particular, all 'local' properties are identical to the 'real' properties. Such an object can be used as a basis for manipulations.

Unlike the name suggests, this node will still contain a hierarchy of nodes below if the original node did so.

has_children?

Signature: *[const]* bool **has_children?**

Description: Test, if there are children

id

Signature: *[const]* unsigned int **id**

Description: Obtain the unique ID

Each layer properties node object has a unique ID that is created when a new LayerPropertiesNode object is instantiated. The ID is copied when the object is copied. The ID can be used to identify the object irregardless of its content.

is_expanded?

Signature: *[const]* bool **is_expanded?**

Description: Gets a value indicating whether the layer tree node is expanded.

This predicate has been introduced in version 0.28.6.

Python specific notes:
The object exposes a readable attribute 'expanded'. This is the getter.

list_index

Signature: *[const]* unsigned int **list_index**

Description: Gets the index of the layer properties list that the node lives in

view

Signature: LayoutView ptr **view**

Description: Gets the view this node lives in

This reference can be nil if the node is a orphan node that lives outside a view.

4.197. API reference - Class LayerPropertiesNodeRef

Notation used in Ruby API documentation

Module: lay

Description: A class representing a reference to a layer properties node

Class hierarchy: LayerPropertiesNodeRef » LayerPropertiesNode » LayerProperties

This object is returned by the layer properties iterator's current method (LayerPropertiesIterator#current). A reference behaves like a layer properties node, but changes in the node are reflected in the view it is attached to.

A typical use case for references is this:

```
# Hides a layers of a view
view = RBA::LayoutView::current
view.each_layer do |lref|
  # lref is a LayerPropertiesNodeRef object
  lref.visible = false
end
```

This class has been introduced in version 0.25.

Public methods

	void	**_assign**	(const LayerProperties other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new LayerPropertiesNodeRef ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const LayerProperties other)	Assigns the contents of the 'other' object to self.
	void	**assign**	(const LayerPropertiesNode other)	Assigns the contents of the 'other' object to self.
	void	**delete**		Erases the current node and all child nodes

[const]	LayerPropertiesNode	**dup**	Creates a LayerPropertiesNode object as a copy of the content of this node.
[const]	bool	**is_valid?**	Returns true, if the reference points to a valid layer properties node

Detailed description

_assign

Signature: void **_assign** (const LayerPropertiesNodeRef other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new LayerPropertiesNodeRef ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it

is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

(1) Signature: void **assign** (const LayerPropertiesNode other)

Description: Assigns the contents of the 'other' object to self.

This version accepts a LayerPropertiesNode object and allows modification of the layer node's hierarchy. Assignment will reconfigure the layer node in the view.

(2) Signature: void **assign** (const LayerProperties other)

Description: Assigns the contents of the 'other' object to self.

This version accepts a LayerProperties object. Assignment will change the properties of the layer in the view.

delete

Signature: void **delete**

Description: Erases the current node and all child nodes

After erasing the node, the reference will become invalid.

dup

Signature: *[const]* LayerPropertiesNode **dup**

Description: Creates a LayerPropertiesNode object as a copy of the content of this node.

This method is mainly provided for backward compatibility with 0.24 and before.

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_valid?

Signature: *[const]* bool **is_valid?**

Description: Returns true, if the reference points to a valid layer properties node

Invalid references behave like ordinary LayerPropertiesNode objects but without the ability to update the view upon changes of attributes.

4.198. API reference - Class LayerPropertiesIterator

Notation used in Ruby API documentation

Module: lay

Description: Layer properties iterator

This iterator provides a flat view for the layers in the layer tree if used with the next method. In this mode it will descend into the hierarchy and deliver node by node as a linear (flat) sequence.

The iterator can also be used to navigate through the node hierarchy using next_sibling, down_first_child, parent etc.

The iterator also plays an important role for manipulating the layer properties tree, i.e. by specifying insertion points in the tree for the LayoutView class.

Public constructors

new LayerPropertiesIterator ptr	**new**	Creates a new object of this class

Public methods

[const]	bool	**!=**	(const LayerPropertie other)	Inequality
[const]	bool	**<**	(const LayerPropertiesIterator other)	Comparison
[const]	bool	**==**	(const LayerPropertie other)	Equality
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const LayerPropertiesIterator other)	Assigns another object to self
[const]	bool	**at_end?**		At-the-end property
[const]	bool	**at_top?**		At-the-top property

[const]	unsigned long	**child_index**		Returns the index of the child within the parent
[const]	LayerPropertiesNodeRef	**current**		Returns a reference to the layer properties node that the iterator points to
	LayerPropertiesIterator	**down_first_child**		Move to the first child
	LayerPropertiesIterator	**down_last_child**		Move to the last child
[const]	new LayerPropertiesIterator ptr	**dup**		Creates a copy of self
[const]	LayerPropertiesIterator	**first_child**		Returns the iterator pointing to the first child
[const]	bool	**is_null?**		"is null" predicate
[const]	LayerPropertiesIterator	**last_child**		Returns the iterator pointing behind the last child
	LayerPropertiesIterator	**next**		Increment operator
	LayerPropertiesIterator	**next_sibling**	(long n)	Move to the next sibling by a given distance
[const]	unsigned long	**num_siblings**		Return the number of siblings
[const]	LayerPropertiesIterator	**parent**		Returns the iterator pointing to the parent node
	LayerPropertiesIterator	**to_sibling**	(unsigned long n)	Move to the sibling with the given index
	LayerPropertiesIterator	**up**		Move up

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const LayerPropertiesIterator other)

Description: Inequality

other: The other object to compare against

<

Signature: *[const]* bool **<** (const LayerPropertiesIterator other)

Description: Comparison

other: The other object to compare against

Returns: true, if self points to an object that comes before other

==

Signature: *[const]* bool **==** (const LayerPropertiesIterator other)

Description: Equality

other: The other object to compare against

Returns true, if self and other point to the same layer properties node. Caution: this does not imply that both layer properties nodes sit in the same tab. Just their position in the tree is compared.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the

reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const LayerPropertiesIterator other)

Description: Assigns another object to self

at_end?

Signature: *[const]* bool **at_end?**

Description: At-the-end property

This predicate is true if the iterator is at the end of either all elements or at the end of the child list (if down_last_child or down_first_child is used to iterate).

at_top?

Signature: *[const]* bool **at_top?**

Description: At-the-top property

This predicate is true if there is no parent node above the node addressed by self.

child_index

Signature: *[const]* unsigned long **child_index**

Description: Returns the index of the child within the parent

This method returns the index of that the properties node the iterator points to in the list of children of its parent. If the element does not have a parent, the index of the element in the global list is returned.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

current

Signature: *[const]* LayerPropertiesNodeRef **current**

Description: Returns a reference to the layer properties node that the iterator points to

Starting with version 0.25, the returned object can be manipulated and the changes will be reflected in the view immediately.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

down_first_child

Signature: LayerPropertiesIterator **down_first_child**

Description: Move to the first child

This method moves to the first child of the current element. If there is no child, at_end? will be true. Even then, the iterator is sitting at the the child level and up can be used to move back.

down_last_child

Signature: LayerPropertiesIterator **down_last_child**

Description: Move to the last child

This method moves behind the last child of the current element. at_end? will be true then. Even then, the iterator points to the child level and up can be used to move back.

Despite the name, the iterator does not address the last child, but the position after that child. To actually get the iterator for the last child, use down_last_child and next_sibling(-1).

dup

Signature: *[const]* new LayerPropertiesIterator ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

first_child

Signature: *[const]* LayerPropertiesIterator **first_child**

Description: Returns the iterator pointing to the first child

If there is no children, the iterator will be a valid insert point but not point to any valid element. It will report at_end? = true.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_null?

Signature: *[const]* bool **is_null?**

Description: "is null" predicate

This predicate is true if the iterator is "null". Such an iterator can be created with the default constructor or by moving a top-level iterator up.

last_child

Signature: *[const]* LayerPropertiesIterator **last_child**

Description: Returns the iterator pointing behind the last child

The iterator will be a valid insert point but not point to any valid element. It will report at_end? = true.

Despite the name, the iterator does not address the last child, but the position after that child. To actually get the iterator for the last child, use last_child and call next_sibling(-1) on that iterator.

new

Signature: *[static]* new LayerPropertiesIterator ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

next

Signature: LayerPropertiesIterator **next**

Description: Increment operator

The iterator will be incremented to point to the next layer entry. It will descend into the hierarchy to address child nodes if there are any.

next_sibling

Signature: LayerPropertiesIterator **next_sibling** (long n)

Description: Move to the next sibling by a given distance

The iterator is moved to the nth next sibling of the current element. Use negative distances to move backward.

num_siblings

Signature: *[const]* unsigned long **num_siblings**

Description: Return the number of siblings

The count includes the current element. More precisely, this property delivers the number of children of the current node's parent.

parent

Signature: *[const]* LayerPropertiesIterator **parent**

Description: Returns the iterator pointing to the parent node

This method will return an iterator pointing to the parent element. If there is no parent, the returned iterator will be a null iterator.

to_sibling

Signature: LayerPropertiesIterator **to_sibling** (unsigned long n)

Description: Move to the sibling with the given index

The iterator is moved to the nth sibling by selecting the nth child in the current node's parent.

up

Signature: LayerPropertiesIterator **up**

Description: Move up

The iterator is moved to point to the current element's parent. If the current element does not have a parent, the iterator will become a null iterator.

4.199. API reference - Class LayoutView::SelectionMode

Notation used in Ruby API documentation

Module: lay

Description: Specifies how selected objects interact with already selected ones.

This class is equivalent to the class LayoutView::SelectionMode

This enum was introduced in version 0.27.

Public constructors

new LayoutView::SelectionMode ptr	**new**	(int i)	Creates an enum from an integer value
new LayoutView::SelectionMode ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const LayoutView::SelectionMc other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const LayoutView::SelectionMc other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const LayoutView::SelectionMc other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	LayoutView::SelectionMode	**Add**	Adds to any existing selection
[static,const]	LayoutView::SelectionMode	**Invert**	Adds to any existing selection, if it's not there yet or removes it from the selection if it's already selected
[static,const]	LayoutView::SelectionMode	**Replace**	Replaces the existing selection

[static,const]	LayoutView::SelectionMode	**Reset**	Removes from any existing selection

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const LayoutView::SelectionMode other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const LayoutView::SelectionMode other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const LayoutView::SelectionMode other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
Add	**Signature**: *[static,const]* LayoutView::SelectionMode **Add** **Description**: Adds to any existing selection **Python specific notes:** The object exposes a readable attribute 'Add'. This is the getter.
Invert	**Signature**: *[static,const]* LayoutView::SelectionMode **Invert** **Description**: Adds to any existing selection, if it's not there yet or removes it from the selection if it's already selected **Python specific notes:** The object exposes a readable attribute 'Invert'. This is the getter.
Replace	**Signature**: *[static,const]* LayoutView::SelectionMode **Replace** **Description**: Replaces the existing selection **Python specific notes:** The object exposes a readable attribute 'Replace'. This is the getter.
Reset	**Signature**: *[static,const]* LayoutView::SelectionMode **Reset** **Description**: Removes from any existing selection **Python specific notes:** The object exposes a readable attribute 'Reset'. This is the getter.
hash	**Signature**: *[const]* int **hash** **Description**: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

new

(1) Signature: *[static]* new LayoutView::SelectionMode ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new LayoutView::SelectionMode ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.200. API reference - Class CellView

Notation used in Ruby API documentation

Module: lay

Description: A class describing what is shown inside a layout view

The cell view points to a specific cell within a certain layout and a hierarchical context. For that, first of all a layout pointer is provided. The cell itself is addressed by an cell_index or a cell object reference. The layout pointer can be nil, indicating that the cell view is invalid.

The cell is not only identified by its index or object but also by the path leading to that cell. This path indicates how to find the cell in the hierarchical context of its parent cells.

The path is in fact composed of two parts: first in an unspecific fashion, just describing which parent cells are used. The target of this path is called the "context cell". It is accessible by the ctx_cell_index or ctx_cell methods. In the viewer, the unspecific part of the path is the location of the cell in the cell tree.

Additionally the path's second part may further identify a specific instance of a certain subcell in the context cell. This is done through a set of InstElement objects. The target of this specific path is the actual cell addressed by the cellview. This target cell is accessible by the cell_index or cell methods. In the viewer, the target cell is shown in the context of the context cell. The hierarchy levels are counted from the context cell, which is on level 0. If the context path is empty, the context cell is identical with the target cell.

Starting with version 0.25, the cellview can be modified directly. This will have an immediate effect on the display. For example, the following code will select a different cell:

```
cv = RBA::CellView::active
cv.cell_name = "TOP2"
```

See The Application API for more details about the cellview objects.

Public constructors

new CellView ptr	**new**	Creates a new object of this class

Public methods

[const]	bool	**==**	(const CellView other)	Equality: indicates whether the cellviews refer to the same one
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**ascend**		Ascends upwards in the hierarchy.

	void	**assign**	(const CellView other)	Assigns another object to self
[const]	Cell ptr	**cell**		Gets the reference to the target cell currently addressed
	void	**cell=**	(Cell ptr cell)	Sets the cell by reference to a Cell object
[const]	unsigned int	**cell_index**		Gets the target cell's index
	void	**cell_index=**	(unsigned int cell_index)	Sets the path to the given cell
[const]	string	**cell_name**		Gets the name of the target cell currently addressed
	void	**cell_name=**	(string cell_name)	Sets the cell by name
	void	**close**		Closes this cell view
[const]	DCplxTrans	**context_dtrans**		Gets the accumulated transformation of the context path in micron unit space
[const]	InstElement[]	**context_path**		Gets the cell's context path
	void	**context_path=**	(InstElement[] path)	Sets the context path explicitly
[const]	ICplxTrans	**context_trans**		Gets the accumulated transformation of the context path
[const]	Cell ptr	**ctx_cell**		Gets the reference to the context cell currently addressed
[const]	unsigned int	**ctx_cell_index**		Gets the context cell's index
	void	**descend**	(InstElement[] path)	Descends further into the hierarchy.
[const]	new CellView ptr	**dup**		Creates a copy of self
[const]	string	**filename**		Gets filename associated with the layout behind the cellview
	void	**hide_cell**	(const Cell ptr cell)	Hides the given cell
[const]	int	**index**		Gets the index of this cellview in the layout view
	bool	**is_cell_hidden?**	(const Cell ptr cell)	Returns true, if the given cell is hidden
[const]	bool	**is_dirty?**		Gets a flag indicating whether the layout needs saving
[const]	bool	**is_valid?**		Returns true, if the cellview is valid

[const]	Layout ptr	**layout**		Gets the reference to the layout object addressed by this view
[const]	string	**name**		Gets the unique name associated with the layout behind the cellview
	void	**name=**	(string name)	sets the unique name associated with the layout behind the cellview
[signal]	void	**on_technology_changed**		An event indicating that the technology has changed
[const]	unsigned int[]	**path**		Gets the cell's unspecific part of the path leading to the context cell
	void	**path=**	(unsigned int[] path)	Sets the unspecific part of the path explicitly
	void	**reset_cell**		Resets the cell
	void	**set_cell**	(unsigned int cell_index)	Sets the path to the given cell
	void	**set_cell_name**	(string cell_name)	Sets the cell by name
	void	**set_context_path**	(InstElement[] path)	Sets the context path explicitly
	void	**set_path**	(unsigned int[] path)	Sets the unspecific part of the path explicitly
	void	**show_all_cells**		Makes all cells shown (cancel effects of hide_cell) for the specified cell view
	void	**show_cell**	(const Cell ptr cell)	Shows the given cell (cancels the effect of hide_cell)
[const]	string	**technology**		Returns the technology name for the layout behind the given cell view
[const]	void	**technology=**	(string tech_name)	Sets the technology for the layout behind the given cell view
	LayoutView ptr	**view**		Gets the view the cellview resides in

Public static methods and constants

CellView	**active**	Gets the active CellView

Deprecated methods (protected, public, static, non-static and constructors)

void	**create**	Use of this method is deprecated. Use _create instead
void	**destroy**	Use of this method is deprecated. Use _destroy instead

[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

==

Signature: *[const]* bool **==** (const CellView other)

Description: Equality: indicates whether the cellviews refer to the same one

In version 0.25, the definition of the equality operator has been changed to reflect identity of the cellview. Before that version, identity of the cell shown was implied.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the

reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

active

Signature: *[static]* CellView **active**

Description: Gets the active CellView

The active CellView is the one that is selected in the current layout view. This method is equivalent to

```
RBA::LayoutView::current.active_cellview
```

If no CellView is active, this method returns nil.

This method has been introduced in version 0.23.

ascend

Signature: void **ascend**

Description: Ascends upwards in the hierarchy.

Removes one element from the specific path of the cellview with the given index. Returns the element removed. This method has been added in version 0.25.

assign

Signature: void **assign** (const CellView other)

Description: Assigns another object to self

cell

Signature: *[const]* Cell ptr **cell**

Description: Gets the reference to the target cell currently addressed

Python specific notes:
The object exposes a readable attribute 'cell'. This is the getter.

cell=

Signature: void **cell=** (Cell ptr cell)

Description: Sets the cell by reference to a Cell object

Setting the cell reference to nil invalidates the cellview. This method will construct any path to this cell, not a particular one. It will clear the context path and update the context and target cell.

Python specific notes:
The object exposes a writable attribute 'cell'. This is the setter.

cell_index

Signature: *[const]* unsigned int **cell_index**

Description: Gets the target cell's index

Python specific notes:
The object exposes a readable attribute 'cell_index'. This is the getter.

cell_index=

Signature: void **cell_index=** (unsigned int cell_index)

Description: Sets the path to the given cell

This method will construct any path to this cell, not a particular one. It will clear the context path and update the context and target cell. Note that the cell is specified by its index.

Python specific notes:
The object exposes a writable attribute 'cell_index'. This is the setter.

cell_name

Signature: *[const]* string **cell_name**

Description: Gets the name of the target cell currently addressed

Python specific notes:
The object exposes a readable attribute 'cell_name'. This is the getter.

cell_name=

Signature: void **cell_name=** (string cell_name)

Description: Sets the cell by name

If the name is not a valid one, the cellview will become invalid. This method will construct any path to this cell, not a particular one. It will clear the context path and update the context and target cell.

Python specific notes:
The object exposes a writable attribute 'cell_name'. This is the setter.

close

Signature: void **close**

Description: Closes this cell view

This method will close the cellview - remove it from the layout view. After this method was called, the cellview will become invalid (see is_valid?).

This method was introduced in version 0.25.

context_dtrans

Signature: *[const]* DCplxTrans **context_dtrans**

Description: Gets the accumulated transformation of the context path in micron unit space

This is the transformation applied to the target cell before it is shown in the context cell Technically this is the product of all transformations over the context path. See context_trans for a version delivering an integer-unit space transformation.

This method has been introduced in version 0.27.3.

context_path

Signature: *[const]* InstElement[] **context_path**

Description: Gets the cell's context path

The context path leads from the context cell to the target cell in a specific fashion, i.e. describing each instance in detail, not just by cell indexes. If the context and target cell are identical, the context path is empty.

Python specific notes:
The object exposes a readable attribute 'context_path'. This is the getter.

context_path=

Signature: void **context_path=** (InstElement[] path)

Description: Sets the context path explicitly

This method assumes that the unspecific part of the path is established already and that the context path starts from the context cell.

Python specific notes:
The object exposes a writable attribute 'context_path'. This is the setter.

context_trans

Signature: *[const]* ICplxTrans **context_trans**

Description: Gets the accumulated transformation of the context path

This is the transformation applied to the target cell before it is shown in the context cell Technically this is the product of all transformations over the context path. See context_dtrans for a version delivering a micron-unit space transformation.

	This method has been introduced in version 0.27.3.
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
ctx_cell	**Signature**: *[const]* Cell ptr **ctx_cell** **Description**: Gets the reference to the context cell currently addressed
ctx_cell_index	**Signature**: *[const]* unsigned int **ctx_cell_index** **Description**: Gets the context cell's index
descend	**Signature**: void **descend** (InstElement[] path) **Description**: Descends further into the hierarchy. Adds the given path (given as an array of InstElement objects) to the specific path of the cellview with the given index. In effect, the cell addressed by the terminal of the new path components can be shown in the context of the upper cells, if the minimum hierarchy level is set to a negative value. The path is assumed to originate from the current cell and contain specific instances sorted from top to bottom. This method has been added in version 0.25.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
dup	**Signature**: *[const]* new CellView ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
filename	**Signature**: *[const]* string **filename** **Description**: Gets filename associated with the layout behind the cellview
hide_cell	**Signature**: void **hide_cell** (const Cell ptr cell) **Description**: Hides the given cell

This method has been added in version 0.25.

index

Signature: *[const]* int **index**

Description: Gets the index of this cellview in the layout view

The index will be negative if the cellview is not a valid one. This method has been added in version 0.25.

is_cell_hidden?

Signature: bool **is_cell_hidden?** (const Cell ptr cell)

Description: Returns true, if the given cell is hidden

This method has been added in version 0.25.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_dirty?

Signature: *[const]* bool **is_dirty?**

Description: Gets a flag indicating whether the layout needs saving

A layout is 'dirty' if it is modified and needs saving. This method returns true in this case.

This method has been introduced in version 0.24.10.

is_valid?

Signature: *[const]* bool **is_valid?**

Description: Returns true, if the cellview is valid

A cellview may become invalid if the corresponding tab is closed for example.

layout

Signature: *[const]* Layout ptr **layout**

Description: Gets the reference to the layout object addressed by this view

name

Signature: *[const]* string **name**

Description: Gets the unique name associated with the layout behind the cellview

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string name)

Description: sets the unique name associated with the layout behind the cellview

this method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'name'. This is the setter.

new

Signature: *[static]* new CellView ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

on_technology_changed

Signature: *[signal]* void **on_technology_changed**

Description: An event indicating that the technology has changed

This event is triggered when the CellView is attached to a different technology.

This event has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'on_technology_changed'. This is the getter.
The object exposes a writable attribute 'on_technology_changed'. This is the setter.

path

Signature: *[const]* unsigned int[] **path**

Description: Gets the cell's unspecific part of the path leading to the context cell

Python specific notes:
The object exposes a readable attribute 'path'. This is the getter.

path=

Signature: void **path=** (unsigned int[] path)

Description: Sets the unspecific part of the path explicitly

Setting the unspecific part of the path will clear the context path component and update the context and target cell.

Python specific notes:
The object exposes a writable attribute 'path'. This is the setter.

reset_cell

Signature: void **reset_cell**

Description: Resets the cell

The cellview will become invalid. The layout object will still be attached to the cellview, but no cell will be selected.

set_cell

Signature: void **set_cell** (unsigned int cell_index)

Description: Sets the path to the given cell

This method will construct any path to this cell, not a particular one. It will clear the context path and update the context and target cell. Note that the cell is specified by its index.

Python specific notes:
The object exposes a writable attribute 'cell_index'. This is the setter.

set_cell_name

Signature: void **set_cell_name** (string cell_name)

Description: Sets the cell by name

If the name is not a valid one, the cellview will become invalid. This method will construct any path to this cell, not a particular one. It will clear the context path and update the context and target cell.

Python specific notes:
The object exposes a writable attribute 'cell_name'. This is the setter.

set_context_path

Signature: void **set_context_path** (InstElement[] path)

Description: Sets the context path explicitly

This method assumes that the unspecific part of the path is established already and that the context path starts from the context cell.

Python specific notes:
The object exposes a writable attribute 'context_path'. This is the setter.

set_path

Signature: void **set_path** (unsigned int[] path)

Description: Sets the unspecific part of the path explicitly

Setting the unspecific part of the path will clear the context path component and update the context and target cell.

Python specific notes:
The object exposes a writable attribute 'path'. This is the setter.

show_all_cells

Signature: void **show_all_cells**

Description: Makes all cells shown (cancel effects of hide_cell) for the specified cell view

This method has been added in version 0.25.

show_cell

Signature: void **show_cell** (const Cell ptr cell)

Description: Shows the given cell (cancels the effect of hide_cell)

This method has been added in version 0.25.

technology

Signature: *[const]* string **technology**

Description: Returns the technology name for the layout behind the given cell view

This method has been added in version 0.23.

Python specific notes:
The object exposes a readable attribute 'technology'. This is the getter.

technology=

Signature: *[const]* void **technology=** (string tech_name)

Description: Sets the technology for the layout behind the given cell view

According to the specification of the technology, new layer properties may be loaded or the net tracer may be reconfigured. If the layout is shown in multiple views, the technology is updated for all views. This method has been added in version 0.22.

Python specific notes:
The object exposes a writable attribute 'technology'. This is the setter.

view

Signature: LayoutView ptr **view**

Description: Gets the view the cellview resides in

This reference will be nil if the cellview is not a valid one. This method has been added in version 0.25.

4.201. API reference - Class Marker

Notation used in Ruby API documentation

Module: lay

Description: The floating-point coordinate marker object

The marker is a visual object that "marks" (highlights) a certain area of the layout, given by a database object. This object accepts database objects with floating-point coordinates in micron values.

Public constructors

new Marker ptr	**new**	(LayoutView ptr view)		Creates a marker

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	unsigned int	**color**		Gets the color of the marker
	void	**color=**	(unsigned int color)	Sets the color of the marker
	void	**dismissable=**	(bool flag)	Sets a value indicating whether the marker can be hidden
[const]	bool	**dismissable?**		Gets a value indicating whether the marker can be hidden
[const]	int	**dither_pattern**		Gets the stipple pattern index
	void	**dither_pattern=**	(int index)	Sets the stipple pattern index
[const]	unsigned int	**frame_color**		Gets the frame color of the marker
	void	**frame_color=**	(unsigned int color)	Sets the frame color of the marker
[const]	int	**halo**		Gets the halo flag
	void	**halo=**	(int halo)	Sets the halo flag
[const]	bool	**has_color?**		Returns a value indicating whether the marker has a specific color

[const]	bool	**has_frame_color?**		Returns a value indicating whether the marker has a specific frame color
[const]	int	**line_style**		Get the line style
	void	**line_style=**	(int index)	Sets the line style
[const]	int	**line_width**		Gets the line width of the marker
	void	**line_width=**	(int width)	Sets the line width of the marker
	void	**reset_color**		Resets the color of the marker
	void	**reset_frame_color**		Resets the frame color of the marker
	void	**set**	(const DBox box)	Sets the box the marker is to display
	void	**set**	(const DText text)	Sets the text the marker is to display
	void	**set**	(const DEdge edge)	Sets the edge the marker is to display
	void	**set**	(const DPath path)	Sets the path the marker is to display
	void	**set**	(const DPolygon polygon)	Sets the polygon the marker is to display
	void	**set_box**	(const DBox box)	Sets the box the marker is to display
	void	**set_edge**	(const DEdge edge)	Sets the edge the marker is to display
	void	**set_path**	(const DPath path)	Sets the path the marker is to display
	void	**set_polygon**	(const DPolygon polygon)	Sets the polygon the marker is to display
	void	**set_text**	(const DText text)	Sets the text the marker is to display
[const]	int	**vertex_size**		Gets the vertex size of the marker
	void	**vertex_size=**	(int size)	Sets the vertex size of the marker

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

color

Signature: *[const]* unsigned int **color**

Description: Gets the color of the marker

This value is valid only if has_color? is true.

Python specific notes:
The object exposes a readable attribute 'color'. This is the getter.

color=

Signature: void **color=** (unsigned int color)

Description: Sets the color of the marker

The color is a 32bit unsigned integer encoding the RGB values in the lower 3 bytes (blue in the lowest significant byte). The color can be reset with reset_color, in which case, the default foreground color is used.

Python specific notes:
The object exposes a writable attribute 'color'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dismissable=

Signature: void **dismissable=** (bool flag)

Description: Sets a value indicating whether the marker can be hidden

Dismissable markers can be hidden setting "View/Show Markers" to "off". The default setting is "false" meaning the marker can't be hidden.

This attribute has been introduced in version 0.25.4.

Python specific notes:
The object exposes a writable attribute 'dismissable'. This is the setter.

dismissable?

Signature: *[const]* bool **dismissable?**

Description: Gets a value indicating whether the marker can be hidden

See dismissable= for a description of this predicate.

Python specific notes:
The object exposes a readable attribute 'dismissable'. This is the getter.

dither_pattern

Signature: *[const]* int **dither_pattern**

Description: Gets the stipple pattern index

See dither_pattern= for a description of the stipple pattern index.

Python specific notes:

The object exposes a readable attribute 'dither_pattern'. This is the getter.

dither_pattern=

Signature: void **dither_pattern=** (int index)

Description: Sets the stipple pattern index

A value of -1 or less than zero indicates that the marker is not filled. Otherwise, the value indicates which pattern to use for filling the marker.

Python specific notes:
The object exposes a writable attribute 'dither_pattern'. This is the setter.

frame_color

Signature: *[const]* unsigned int **frame_color**

Description: Gets the frame color of the marker

This value is valid only if has_frame_color? is true.The set method has been added in version 0.20.

Python specific notes:
The object exposes a readable attribute 'frame_color'. This is the getter.

frame_color=

Signature: void **frame_color=** (unsigned int color)

Description: Sets the frame color of the marker

The color is a 32bit unsigned integer encoding the RGB values in the lower 3 bytes (blue in the lowest significant byte). The color can be reset with reset_frame_color, in which case the fill color is used. The set method has been added in version 0.20.

Python specific notes:
The object exposes a writable attribute 'frame_color'. This is the setter.

halo

Signature: *[const]* int **halo**

Description: Gets the halo flag

See halo= for a description of the halo flag.

Python specific notes:
The object exposes a readable attribute 'halo'. This is the getter.

halo=

Signature: void **halo=** (int halo)

Description: Sets the halo flag

The halo flag is either -1 (for taking the default), 0 to disable the halo or 1 to enable it. If the halo is enabled, a pixel border with the background color is drawn around the marker, the vertices and texts.

Python specific notes:
The object exposes a writable attribute 'halo'. This is the setter.

has_color?

Signature: *[const]* bool **has_color?**

Description: Returns a value indicating whether the marker has a specific color

has_frame_color?

Signature: *[const]* bool **has_frame_color?**

Description: Returns a value indicating whether the marker has a specific frame color

The set method has been added in version 0.20.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

line_style

Signature: *[const]* int **line_style**

Description: Get the line style

See line_style= for a description of the line style index. This method has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'line_style'. This is the getter.

line_style=

Signature: void **line_style=** (int index)

Description: Sets the line style

The line style is given by an index. 0 is solid, 1 is dashed and so forth.

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'line_style'. This is the setter.

line_width

Signature: *[const]* int **line_width**

Description: Gets the line width of the marker

See line_width= for a description of the line width.

Python specific notes:
The object exposes a readable attribute 'line_width'. This is the getter.

line_width=

Signature: void **line_width=** (int width)

Description: Sets the line width of the marker

This is the width of the line drawn for the outline of the marker.

Python specific notes:
The object exposes a writable attribute 'line_width'. This is the setter.

new

Signature: *[static]* new Marker ptr **new** (LayoutView ptr view)

Description: Creates a marker

A marker is always associated with a view, in which it is shown. The view this marker is associated with must be passed to the constructor.

Python specific notes:
This method is the default initializer of the object.

reset_color

Signature: void **reset_color**

Description: Resets the color of the marker

See set_color for a description of the color property of the marker.

reset_frame_color

Signature: void **reset_frame_color**

Description: Resets the frame color of the marker

See set_frame_color for a description of the frame color property of the marker.The set method has been added in version 0.20.

set

(1) Signature: void **set** (const DBox box)

Description: Sets the box the marker is to display

Makes the marker show a box. The box must be given in micron units. If the box is empty, no marker is drawn. The set method has been added in version 0.20.

(2) Signature: void **set** (const DText text)

Description: Sets the text the marker is to display

Makes the marker show a text. The text must be given in micron units. The set method has been added in version 0.20.

(3) Signature: void **set** (const DEdge edge)

Description: Sets the edge the marker is to display

Makes the marker show a edge. The edge must be given in micron units. The set method has been added in version 0.20.

(4) Signature: void **set** (const DPath path)

Description: Sets the path the marker is to display

Makes the marker show a path. The path must be given in micron units. The set method has been added in version 0.20.

(5) Signature: void **set** (const DPolygon polygon)

Description: Sets the polygon the marker is to display

Makes the marker show a polygon. The polygon must be given in micron units. The set method has been added in version 0.20.

set_box

Signature: void **set_box** (const DBox box)

Description: Sets the box the marker is to display

Makes the marker show a box. The box must be given in micron units. If the box is empty, no marker is drawn. The set method has been added in version 0.20.

set_edge

Signature: void **set_edge** (const DEdge edge)

Description: Sets the edge the marker is to display

Makes the marker show a edge. The edge must be given in micron units. The set method has been added in version 0.20.

set_path

Signature: void **set_path** (const DPath path)

Description: Sets the path the marker is to display

Makes the marker show a path. The path must be given in micron units. The set method has been added in version 0.20.

set_polygon

Signature: void **set_polygon** (const DPolygon polygon)

Description: Sets the polygon the marker is to display

	Makes the marker show a polygon. The polygon must be given in micron units. The set method has been added in version 0.20.
set_text	**Signature**: void **set_text** (const DText text) **Description**: Sets the text the marker is to display Makes the marker show a text. The text must be given in micron units. The set method has been added in version 0.20.
vertex_size	**Signature**: *[const]* int **vertex_size** **Description**: Gets the vertex size of the marker See vertex_size= for a description. **Python specific notes:** The object exposes a readable attribute 'vertex_size'. This is the getter.
vertex_size=	**Signature**: void **vertex_size=** (int size) **Description**: Sets the vertex size of the marker This is the size of the rectangles drawn for the vertices object. **Python specific notes:** The object exposes a writable attribute 'vertex_size'. This is the setter.

4.202. API reference - Class AbstractMenu

Notation used in Ruby API documentation

Module: lay

Description: An abstraction for the application menus

The abstract menu is a class that stores a main menu and several popup menus in a generic form such that they can be manipulated and converted into GUI objects.

Each item can be associated with a Action, which delivers a title, enabled/disable state etc. The Action is either provided when new entries are inserted or created upon initialisation.

The abstract menu class provides methods to manipulate the menu structure (the state of the menu items, their title and shortcut key is provided and manipulated through the Action object).

Menu items and submenus are referred to by a "path". The path is a string with this interpretation:

""	is the root
"[<path>.]<name>"	is an element of the submenu given by <path>. If <path> is omitted, this refers to an element in the root
"[<path>.]end"	refers to the item past the last item of the submenu given by <path> or root
"[<path>.]begin"	refers to the first item of the submenu given by <path> or root
"[<path>.]#<n>"	refers to the nth item of the submenu given by <path> or root (n is an integer number)

Menu items can be put into groups. The path strings of each group can be obtained with the "group" method. An item is put into a group by appending ":<group-name>" to the item's name. This specification can be used several times.

Detached menus (i.e. for use in context menus) can be created as virtual top-level submenus with a name of the form "@<name>". A special detached menu is "@toolbar" which represents the tool bar of the main window. Menus are closely related to the Action class. Actions are used to represent selectable items inside menus, provide the title and other configuration settings. Actions also link the menu items with code. See the Action class description for further details.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	Action ptr	**action**	(string path)	Gets the reference to a Action object associated with the given path
	void	**clear_menu**	(string path)	Deletes the children of the item given by the path
	void	**delete_item**	(string path)	Deletes the item given by the path

[const]	string[]	**group**	(string group)	Gets the group members
	void	**insert_item**	(string path, string name, const Action ptr action)	Inserts a new item before the one given by the path
	void	**insert_menu**	(string path, string name, string title)	Inserts a new submenu before the item given by the path
	void	**insert_menu**	(string path, string name, Action ptr action)	Inserts a new submenu before the item given by the path
	void	**insert_separator**	(string path, string name)	Inserts a new separator before the item given by the path
[const]	bool	**is_menu?**	(string path)	Returns true if the item is a menu
[const]	bool	**is_separator?**	(string path)	Returns true if the item is a separator
[const]	bool	**is_valid?**	(string path)	Returns true if the path is a valid one
[const]	string[]	**items**	(string path)	Gets the subitems for a given submenu

Public static methods and constants

	string	**pack_key_binding**	(map<string,string> path_to_keys)	Serializes a key binding definition into a single string
	string	**pack_menu_items_hidden**	(map<string,bool> path_to_visibility)	Serializes a menu item visibility definition into a single string
	map<string,string>	**unpack_key_binding**	(string s)	Deserializes a key binding definition
	map<string,bool>	**unpack_menu_items_hidden**	(string s)	Deserializes a menu item visibility definition

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

action

Signature: Action ptr **action** (string path)

Description: Gets the reference to a Action object associated with the given path

path:	The path to the item.
Returns:	A reference to a Action object associated with this path or nil if the path is not valid

clear_menu

Signature: void **clear_menu** (string path)

Description: Deletes the children of the item given by the path

path:	The path to the item whose children to delete

This method has been introduced in version 0.28.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

delete_item

Signature: void **delete_item** (string path)

Description: Deletes the item given by the path

path:	The path to the item to delete

This method will also delete all children of the given item. To clear the children only, use clear_menu.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

group

Signature: *[const]* string[] **group** (string group)

Description: Gets the group members

group:	The group name
A:	vector of all members (by path) of the group

insert_item

Signature: void **insert_item** (string path, string name, const Action ptr action)

Description: Inserts a new item before the one given by the path

path:	The path to the item before which to insert the new item
name:	The name of the item to insert
action:	The Action object to insert

The Action object passed as the third parameter references the handler which both implements the action to perform and the menu item's appearance such as title, icon and keyboard shortcut.

insert_menu

(1) Signature: void **insert_menu** (string path, string name, string title)

Description: Inserts a new submenu before the item given by the path

path:	The path to the item before which to insert the submenu
name:	The name of the submenu to insert

title: The title of the submenu to insert

The title string optionally encodes the key shortcut and icon resource in the form <text>["("<shortcut>")"]["<"<icon-resource>">"].

(2) Signature: void **insert_menu** (string path, string name, Action ptr action)

Description: Inserts a new submenu before the item given by the path

path: The path to the item before which to insert the submenu

name: The name of the submenu to insert

action: The action object of the submenu to insert

This method variant has been added in version 0.28.

insert_separator

Signature: void **insert_separator** (string path, string name)

Description: Inserts a new separator before the item given by the path

path: The path to the item before which to insert the separator

name: The name of the separator to insert

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_menu?

Signature: *[const]* bool **is_menu?** (string path)

Description: Returns true if the item is a menu

path: The path to the item

Returns: false if the path is not valid or is not a menu

is_separator?

Signature: *[const]* bool **is_separator?** (string path)

Description: Returns true if the item is a separator

path: The path to the item

Returns: false if the path is not valid or is not a separator

This method has been introduced in version 0.19.

is_valid?

Signature: *[const]* bool **is_valid?** (string path)

Description: Returns true if the path is a valid one

path: The path to check

Returns: false if the path is not a valid path to an item

items

Signature: *[const]* string[] **items** (string path)

Description: Gets the subitems for a given submenu

path: The path to the submenu

	Returns:	A vector or path strings for the child items or an empty vector if the path is not valid or the item does not have children

pack_key_binding

Signature: *[static]* string **pack_key_binding** (map<string,string> path_to_keys)

Description: Serializes a key binding definition into a single string

The serialized format is used by the 'key-bindings' config key. This method will take an array of path/key definitions (including the Action#NoKeyBound option) and convert it to a single string suitable for assigning to the config key.

This method has been introduced in version 0.26.

pack_menu_items_hidden

Signature: *[static]* string **pack_menu_items_hidden** (map<string,bool> path_to_visibility)

Description: Serializes a menu item visibility definition into a single string

The serialized format is used by the 'menu-items-hidden' config key. This method will take an array of path/visibility flag definitions and convert it to a single string suitable for assigning to the config key.

This method has been introduced in version 0.26.

unpack_key_binding

Signature: *[static]* map<string,string> **unpack_key_binding** (string s)

Description: Deserializes a key binding definition

This method is the reverse of pack_key_binding.

This method has been introduced in version 0.26.

unpack_menu_items_hidden

Signature: *[static]* map<string,bool> **unpack_menu_items_hidden** (string s)

Description: Deserializes a menu item visibility definition

This method is the reverse of pack_menu_items_hidden.

This method has been introduced in version 0.26.

4.203. API reference - Class Action

Notation used in Ruby API documentation

Module: lay

Description: The abstraction for an action (i.e. used inside menus)

Class hierarchy: Action

Actions act as a generalization of menu entries. The action provides the appearance of a menu entry such as title, key shortcut etc. and dispatches the menu events. The action can be manipulated to change to appearance of a menu entry and can be attached an observer that receives the events when the menu item is selected.

Multiple action objects can refer to the same action internally, in which case the information and event handler is copied between the incarnations. This way, a single implementation can be provided for multiple places where an action appears, for example inside the toolbar and in addition as a menu entry. Both actions will shared the same icon, text, shortcut etc.

Actions are mainly used for providing new menu items inside the AbstractMenu class. This is some sample Ruby code for that case:

```
a = RBA::Action.new
a.title = "Push Me!"
a.on_triggered do
  puts "I was pushed!"
end

app = RBA::Application.instance
mw = app.main_window

menu = mw.menu
menu.insert_separator("@toolbar.end", "name")
menu.insert_item("@toolbar.end", "my_action", a)
```

This code will register a custom action in the toolbar. When the toolbar button is pushed a message is printed. The toolbar is addressed by a path starting with the pseudo root "@toolbar".

In Version 0.23, the Action class has been merged with the ActionBase class.

Public constructors

new Action ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.

	void	**checkable=**	(bool checkable)	Makes the item(s) checkable or not
	void	**checked=**	(bool checked)	Checks or unchecks the item
[const]	string	**default_shortcut**		Gets the default keyboard shortcut
	void	**default_shortcut=**	(string shortcut)	Sets the default keyboard shortcut
[const]	string	**effective_shortcut**		Gets the effective keyboard shortcut
	void	**enabled=**	(bool enabled)	Enables or disables the action
	void	**hidden=**	(bool hidden)	Sets a value that makes the item hidden always
	void	**icon=**	(string file)	Sets the icon to the given image file
	void	**icon=**	(const QIcon qicon)	Sets the icon to the given QIcon object
[const]	string	**icon_text**		Gets the icon's text
	void	**icon_text=**	(string icon_text)	Sets the icon's text
[const]	bool	**is_checkable?**		Gets a value indicating whether the item is checkable
[const]	bool	**is_checked?**		Gets a value indicating whether the item is checked
[const]	bool	**is_effective_enabled?**		Gets a value indicating whether the item is really enabled
[const]	bool	**is_effective_visible?**		Gets a value indicating whether the item is really visible
[const]	bool	**is_enabled?**		Gets a value indicating whether the item is enabled
[const]	bool	**is_hidden?**		Gets a value indicating whether the item is hidden
[const]	bool	**is_separator?**		Gets a value indicating whether the item is a separator
[const]	bool	**is_visible?**		Gets a value indicating whether the item is visible
[const]	Macro ptr	**macro**		Gets the macro associated with the action
[virtual]	void	**menu_opening**		This method is called if the menu item is a sub-menu and before the menu is opened.

[signal]	void	**on_menu_opening**		This event is called if the menu item is a sub-menu and before the menu is opened.
[signal]	void	**on_triggered**		This event is called if the menu item is selected.
	void	**separator=**	(bool separator)	Makes an item a separator or not
[const]	string	**shortcut**		Gets the keyboard shortcut
	void	**shortcut=**	(string shortcut)	Sets the keyboard shortcut
[const]	string	**title**		Gets the title
	void	**title=**	(string title)	Sets the title
[const]	string	**tool_tip**		Gets the tool tip text.
	void	**tool_tip=**	(string text)	Sets the tool tip text
	void	**trigger**		Triggers the action programmatically
[virtual]	void	**triggered**		This method is called if the menu item is selected.
	void	**visible=**	(bool visible)	Sets the item's visibility
[virtual,const]	bool	**wants_enabled**		Returns a value whether the action wants to become enabled.
[virtual,const]	bool	**wants_visible**		Returns a value whether the action wants to become visible

Public static methods and constants

[static,const]	string	**NoKeyBound**	Gets a shortcut value indicating that no shortcut shall be assigned

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

NoKeyBound

Signature: *[static,const]* string **NoKeyBound**

Description: Gets a shortcut value indicating that no shortcut shall be assigned

This method has been introduced in version 0.26.

Python specific notes:
The object exposes a readable attribute 'NoKeyBound'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

checkable=

Signature: void **checkable=** (bool checkable)

Description: Makes the item(s) checkable or not

checkable: true to make the item checkable

Python specific notes:
The object exposes a writable attribute 'checkable'. This is the setter.

checked=

Signature: void **checked=** (bool checked)

Description: Checks or unchecks the item

checked: true to make the item checked

Python specific notes:
The object exposes a writable attribute 'checked'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

default_shortcut

Signature: *[const]* string **default_shortcut**

Description: Gets the default keyboard shortcut

Returns: The default keyboard shortcut as a string

This attribute has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'default_shortcut'. This is the getter.

default_shortcut=

Signature: void **default_shortcut=** (string shortcut)

Description: Sets the default keyboard shortcut

The default shortcut is used, if shortcut is empty.

This attribute has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'default_shortcut'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

effective_shortcut

Signature: *[const]* string **effective_shortcut**

Description: Gets the effective keyboard shortcut

Returns: The effective keyboard shortcut as a string

The effective shortcut is the one that is taken. It's either shortcut or default_shortcut.

This attribute has been introduced in version 0.25.

enabled=

Signature: void **enabled=** (bool enabled)

Description: Enables or disables the action

enabled: true to enable the item

Python specific notes:
The object exposes a writable attribute 'enabled'. This is the setter.

hidden=

Signature: void **hidden=** (bool hidden)

Description: Sets a value that makes the item hidden always

See is_hidden? for details.

This attribute has been introduced in version 0.25

Python specific notes:
The object exposes a writable attribute 'hidden'. This is the setter.

icon=

(1) Signature: void **icon=** (string file)

Description: Sets the icon to the given image file

file: The image file to load for the icon

Passing an empty string will reset the icon.

Python specific notes:
The object exposes a writable attribute 'icon'. This is the setter.

(2) Signature: void **icon=** (const QIcon qicon)

Description: Sets the icon to the given QIcon object

qicon: The QIcon object

This variant has been added in version 0.28.

Python specific notes:
The object exposes a writable attribute 'icon'. This is the setter.

icon_text

Signature: *[const]* string **icon_text**

Description: Gets the icon's text

Python specific notes:
The object exposes a readable attribute 'icon_text'. This is the getter.

icon_text=

Signature: void **icon_text=** (string icon_text)

Description: Sets the icon's text

If an icon text is set, this will be used for the text below the icon. If no icon text is set, the normal text will be used for the icon. Passing an empty string will reset the icon's text.

Python specific notes:

	The object exposes a writable attribute 'icon_text'. This is the setter.
is_checkable?	**Signature**: *[const]* bool **is_checkable?** **Description**: Gets a value indicating whether the item is checkable **Python specific notes:** The object exposes a readable attribute 'checkable'. This is the getter.
is_checked?	**Signature**: *[const]* bool **is_checked?** **Description**: Gets a value indicating whether the item is checked **Python specific notes:** The object exposes a readable attribute 'checked'. This is the getter.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
is_effective_enabled?	**Signature**: *[const]* bool **is_effective_enabled?** **Description**: Gets a value indicating whether the item is really enabled This is the combined value from is_enabled? and dynamic value (wants_enabled). This attribute has been introduced in version 0.28.
is_effective_visible?	**Signature**: *[const]* bool **is_effective_visible?** **Description**: Gets a value indicating whether the item is really visible This is the combined visibility from is_visible? and is_hidden? and dynamic visibility (wants_visible). This attribute has been introduced in version 0.25.
is_enabled?	**Signature**: *[const]* bool **is_enabled?** **Description**: Gets a value indicating whether the item is enabled **Python specific notes:** The object exposes a readable attribute 'enabled'. This is the getter.
is_hidden?	**Signature**: *[const]* bool **is_hidden?** **Description**: Gets a value indicating whether the item is hidden If an item is hidden, it's always hidden and is_visible? does not have an effect. This attribute has been introduced in version 0.25. **Python specific notes:** The object exposes a readable attribute 'hidden'. This is the getter.
is_separator?	**Signature**: *[const]* bool **is_separator?** **Description**: Gets a value indicating whether the item is a separator This method has been introduced in version 0.25. **Python specific notes:** The object exposes a readable attribute 'separator'. This is the getter.

is_visible?

Signature: *[const]* bool **is_visible?**

Description: Gets a value indicating whether the item is visible

The visibility combines with is_hidden?. To get the true visiblity, use is_effective_visible?.

Python specific notes:
The object exposes a readable attribute 'visible'. This is the getter.

macro

Signature: *[const]* Macro ptr **macro**

Description: Gets the macro associated with the action

If the action is associated with a macro, this method returns a reference to the Macro object. Otherwise, this method returns nil.

This method has been added in version 0.25.

menu_opening

Signature: *[virtual]* void **menu_opening**

Description: This method is called if the menu item is a sub-menu and before the menu is opened.

Reimplement this method is a derived class to receive this event. You can also use the on_menu_opening event instead.

This method has been added in version 0.28.

new

Signature: *[static]* new Action ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

on_menu_opening

Signature: *[signal]* void **on_menu_opening**

Description: This event is called if the menu item is a sub-menu and before the menu is opened.

This event provides an opportunity to populate the menu before it is opened.

This event has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'on_menu_opening'. This is the getter.
The object exposes a writable attribute 'on_menu_opening'. This is the setter.

on_triggered

Signature: *[signal]* void **on_triggered**

Description: This event is called if the menu item is selected.

This event has been introduced in version 0.21 and moved to the ActionBase class in 0.28.

Python specific notes:
The object exposes a readable attribute 'on_triggered'. This is the getter.
The object exposes a writable attribute 'on_triggered'. This is the setter.

separator=

Signature: void **separator=** (bool separator)

Description: Makes an item a separator or not

separator: true to make the item a separator

This method has been introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'separator'. This is the setter.

shortcut

Signature: *[const]* string **shortcut**

Description: Gets the keyboard shortcut

Returns: The keyboard shortcut as a string

Python specific notes:
The object exposes a readable attribute 'shortcut'. This is the getter.

shortcut=

Signature: void **shortcut=** (string shortcut)

Description: Sets the keyboard shortcut

shortcut: The keyboard shortcut in Qt notation (i.e. "Ctrl+C")

If the shortcut string is empty, the default shortcut will be used. If the string is equal to Action#NoKeyBound, no keyboard shortcut will be assigned.

The NoKeyBound option has been added in version 0.26.

Python specific notes:
The object exposes a writable attribute 'shortcut'. This is the setter.

title

Signature: *[const]* string **title**

Description: Gets the title

Returns: The current title string

Python specific notes:
The object exposes a readable attribute 'title'. This is the getter.

title=

Signature: void **title=** (string title)

Description: Sets the title

title: The title string to set (just the title)

Python specific notes:
The object exposes a writable attribute 'title'. This is the setter.

tool_tip

Signature: *[const]* string **tool_tip**

Description: Gets the tool tip text.

This method has been added in version 0.22.

Python specific notes:
The object exposes a readable attribute 'tool_tip'. This is the getter.

tool_tip=

Signature: void **tool_tip=** (string text)

Description: Sets the tool tip text

The tool tip text is displayed in the tool tip window of the menu entry. This is in particular useful for entries in the tool bar. This method has been added in version 0.22.

Python specific notes:
The object exposes a writable attribute 'tool_tip'. This is the setter.

trigger

Signature: void **trigger**

Description: Triggers the action programmatically

triggered

Signature: *[virtual]* void **triggered**

Description: This method is called if the menu item is selected.

Reimplement this method is a derived class to receive this event. You can also use the on_triggered event instead.

visible=

Signature: void **visible=** (bool visible)

Description: Sets the item's visibility

visible: true to make the item visible

Python specific notes:
The object exposes a writable attribute 'visible'. This is the setter.

wants_enabled

Signature: *[virtual,const]* bool **wants_enabled**

Description: Returns a value whether the action wants to become enabled.

This is a dynamic query for enabled state which the system uses to dynamically show or hide menu items. This information is evaluated in addition to is_enabled? and contributes to the effective enabled status from is_effective_enabled?.

This feature has been introduced in version 0.28.

wants_visible

Signature: *[virtual,const]* bool **wants_visible**

Description: Returns a value whether the action wants to become visible

This is a dynamic query for visibility which the system uses to dynamically show or hide menu items, for example in the MRU lists. This visibility information is evaluated in addition to is_visible? and is_hidden? and contributes to the effective visibility status from is_effective_visible?.

This feature has been introduced in version 0.28.

4.204. API reference - Class PluginFactory

Notation used in Ruby API documentation

Module: lay

Description: The plugin framework's plugin factory object

Plugins are components that extend KLayout's functionality in various aspects. Scripting support exists currently for providing mouse mode handlers and general on-demand functionality connected with a menu entry.

Plugins are objects that implement the Plugin interface. Each layout view is associated with one instance of such an object. The PluginFactory is a singleton which is responsible for creating Plugin objects and providing certain configuration information such as where to put the menu items connected to this plugin and what configuration keys are used.

An implementation of PluginFactory must at least provide an implementation of create_plugin. This method must instantiate a new object of the specific plugin.

After the factory has been created, it must be registered in the system using one of the register methods. It is therefore recommended to put the call to register at the end of the "initialize" method. For the registration to work properly, the menu items must be defined before register is called.

The following features can also be implemented:

- Reserve keys in the configuration file using add_option in the constructor
- Create menu items by using add_menu_entry in the constructor
- Set the title for the mode entry that appears in the tool bar using the register argument
- Provide global functionality (independent from the layout view) using configure or menu_activated

This is a simple example for a plugin in Ruby. It switches the mouse cursor to a 'cross' cursor when it is active:

```
class PluginTestFactory < RBA::PluginFactory

  # Constructor
  def initialize
    # registers the new plugin class at position 100000 (at the end), with name
    # "my_plugin_test" and title "My plugin test"
    register(100000, "my_plugin_test", "My plugin test")
  end

  # Create a new plugin instance of the custom type
  def create_plugin(manager, dispatcher, view)
    return PluginTest.new
  end

end

# The plugin class
class PluginTest < RBA::Plugin
  def mouse_moved_event(p, buttons, prio)
    if prio
      # Set the cursor to cross if our plugin is active.
      set_cursor(RBA::Cursor::Cross)
    end
    # Returning false indicates that we don't want to consume the event.
    # This way for example the cursor position tracker still works.
    false
  end
  def mouse_click_event(p, buttons, prio)
    if prio
      puts "mouse button clicked."
      # This indicates we want to consume the event and others don't receive the mouse click
      # with prio = false.
```

```
      return true
    end
    # don't consume the event if we are not active.
    false
  end
end

# Instantiate the new plugin factory.
PluginTestFactory.new
```

This class has been introduced in version 0.22.

Public constructors

new PluginFactory ptr	new	Creates a new object of this class

Public methods

	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
	void	add_config_menu_iter	(string menu_name, string insert_pos, string title, string cname, string cvalue)	Adds a configuration menu item
	void	add_menu_entry	(string menu_name, string insert_pos)	Specifies a separator
	void	add_menu_entry	(string symbol, string menu_name, string insert_pos, string title)	Specifies a menu item
	void	add_menu_item_clone	(string symbol, string menu_name, string insert_pos, string copy_from)	Specifies a menu item as a clone of another one
	void	add_option	(string name, string default_value)	Specifies configuration variables.
	void	add_submenu	(string menu_name, string insert_pos,	Specifies a menu item or sub-menu

			string title)	
[virtual]	void	**config_finalize**		Gets called after a set of configuration events has been sent
[virtual]	bool	**configure**	(string name, string value)	Gets called for configuration events for the plugin singleton
[virtual,const	new Plugin ptr	**create_plugin**	(Manager ptr manager, Dispatcher ptr dispatcher, LayoutView ptr view)	Creates the plugin
	void	**has_tool_entry=**	(bool f)	Enables or disables the tool bar entry
[virtual]	void	**initialized**	(Dispatcher ptr dispatcher)	Gets called when the plugin singleton is initialized, i.e. when the application has been started.
[virtual,const]	bool	**menu_activated**	(string symbol)	Gets called when a menu item is selected
	void	**register**	(int position, string name, string title)	Registers the plugin factory
	void	**register**	(int position, string name, string title, string icon)	Registers the plugin factory
[virtual]	void	**uninitialized**	(Dispatcher ptr dispatcher)	Gets called when the application shuts down and the plugin is unregistered

Deprecated methods (protected, public, static, non-static and constructors)

	void	**add_menu_entry**	(string symbol, string menu_name, string insert_pos, string title, bool sub_menu)	Use of this method is deprecated
	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_config_menu_item

Signature: void **add_config_menu_item** (string menu_name, string insert_pos, string title, string cname, string cvalue)

Description: Adds a configuration menu item

Menu items created this way will send a configuration request with 'cname' as the configuration parameter name and 'cvalue' as the configuration parameter value.

This method has been introduced in version 0.27.

add_menu_entry

(1) Signature: void **add_menu_entry** (string menu_name, string insert_pos)

Description: Specifies a separator

Call this method in the factory constructor to build the menu items that this plugin shall create. This specific call inserts a separator at the given position (insert_pos). The position uses abstract menu item paths and "menu_name" names the component that will be created. See AbstractMenu for a description of the path.

(2) Signature: void **add_menu_entry** (string symbol, string menu_name, string insert_pos, string title)

Description: Specifies a menu item

symbol:	The string to send to the plugin if the menu is triggered
menu_name:	The name of entry to create at the given position
insert_pos:	The position where to create the entry
title:	The title string for the item. The title can contain a keyboard shortcut in round braces after the title text, i.e. "My Menu Item(F12)"

Call this method in the factory constructor to build the menu items that this plugin shall create. This specific call inserts a menu item at the specified position (insert_pos). The position uses abstract menu item paths and "menu_name" names the component that will be created. See AbstractMenu for a description of the path. When the menu item is selected "symbol" is the string that is sent to the menu_activated callback (either the global one for the factory ot the one of the per-view plugin instance).

(3) Signature: void **add_menu_entry** (string symbol, string menu_name, string insert_pos, string title, bool sub_menu)

Description: Specifies a menu item or sub-menu

Use of this method is deprecated

Similar to the previous form of "add_menu_entry", but this version allows also to create sub-menus by setting the last parameter to "true".

With version 0.27 it's more convenient to use add_submenu.

add_menu_item_clone

Signature: void **add_menu_item_clone** (string symbol, string menu_name, string insert_pos, string copy_from)

Description: Specifies a menu item as a clone of another one

Using this method, a menu item can be made a clone of another entry (given as path by 'copy_from'). The new item will share the Action object with the original one, so manipulating the action will change both the original entry and the new entry.

This method has been introduced in version 0.27.

add_option

Signature: void **add_option** (string name, string default_value)

Description: Specifies configuration variables.

Call this method in the factory constructor to add configuration key/value pairs to the configuration repository. Without specifying configuration variables, the status of a plugin cannot be persisted.

Once the configuration variables are known, they can be retrieved on demand using "get_config" from MainWindow or listening to configure callbacks (either in the factory or the plugin instance). Configuration variables can be set using "set_config" from MainWindow. This scheme also works

without registering the configuration options, but doing so has the advantage that it is guaranteed that a variable with this keys exists and has the given default value initially.

add_submenu

Signature: void **add_submenu** (string menu_name, string insert_pos, string title)

Description: Specifies a menu item or sub-menu

This method has been introduced in version 0.27.

config_finalize

Signature: *[virtual]* void **config_finalize**

Description: Gets called after a set of configuration events has been sent

This method can be reimplemented and is called after a set of configuration events has been sent to the plugin factory singleton with configure. It can be used to set up user interfaces properly for example.

configure

Signature: *[virtual]* bool **configure** (string name, string value)

Description: Gets called for configuration events for the plugin singleton

name:	The configuration key
value:	The value of the configuration variable
Returns:	True to stop further processing

This method can be reimplemented to receive configuration events for the plugin singleton. Before a configuration can be received it must be registered by calling add_option in the plugin factories' constructor.

The implementation of this method may return true indicating that the configuration request will not be handled by further modules. It's more cooperative to return false which will make the system distribute the configuration request to other receivers as well.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

create_plugin

Signature: *[virtual,const]* new Plugin ptr **create_plugin** (Manager ptr manager, Dispatcher ptr dispatcher, LayoutView ptr view)

Description: Creates the plugin

manager:	The database manager object responsible for handling database transactions
dispatcher:	The reference to the MainWindow object
view:	The LayoutView that is plugin is created for
Returns:	The new Plugin implementation object

This is the basic functionality that the factory must provide. This method must create a plugin of the specific type.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

has_tool_entry=

Signature: void **has_tool_entry=** (bool f)

Description: Enables or disables the tool bar entry

Initially this property is set to true. This means that the plugin will have a visible entry in the toolbar. This property can be set to false to disable this feature. In that case, the title and icon given on registration will be ignored.

Python specific notes:
The object exposes a writable attribute 'has_tool_entry'. This is the setter.

initialized

Signature: *[virtual]* void **initialized** (Dispatcher ptr dispatcher)

Description: Gets called when the plugin singleton is initialized, i.e. when the application has been started.

dispatcher: The reference to the MainWindow object

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

menu_activated

Signature: *[virtual,const]* bool **menu_activated** (string symbol)

Description: Gets called when a menu item is selected

Usually, menu-triggered functionality is implemented in the per-view instance of the plugin. However, using this method it is possible to implement functionality globally for all plugin instances. The symbol is the string registered with the specific menu item in the add_menu_item call.

If this method was handling the menu event, it should return true. This indicates that the event will not be propagated to other plugins hence avoiding duplicate calls.

new

Signature: *[static]* new PluginFactory ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

register

(1) Signature: void **register** (int position, string name, string title)

Description: Registers the plugin factory

position: An integer that determines the order in which the plugins are created. The internal plugins use the values from 1000 to 50000.

name: The plugin name. This is an arbitrary string which should be unique. Hence it is recommended to use a unique prefix, i.e. "myplugin::ThePluginClass".

title: The title string which is supposed to appear in the tool bar and menu related to this plugin.

Registration of the plugin factory makes the object known to the system. Registration requires that the menu items have been set already. Hence it is recommended to put the registration at the end of the initialization method of the factory class.

(2) Signature: void **register** (int position, string name, string title, string icon)

Description: Registers the plugin factory

position: An integer that determines the order in which the plugins are created. The internal plugins use the values from 1000 to 50000.

name: The plugin name. This is an arbitrary string which should be unique. Hence it is recommended to use a unique prefix, i.e. "myplugin::ThePluginClass".

title: The title string which is supposed to appear in the tool bar and menu related to this plugin.

icon: The path to the icon that appears in the tool bar and menu related to this plugin.

This version also allows registering an icon for the tool bar.

Registration of the plugin factory makes the object known to the system. Registration requires that the menu items have been set already. Hence it is recommended to put the registration at the end of the initialization method of the factory class.

uninitialized

Signature: *[virtual]* void **uninitialized** (Dispatcher ptr dispatcher)

Description: Gets called when the application shuts down and the plugin is unregistered

dispatcher: The reference to the MainWindow object

This event can be used to free resources allocated with this factory singleton.

4.205. API reference - Class Plugin

Notation used in Ruby API documentation

Module: lay

Description: The plugin object

This class provides the actual plugin implementation. Each view gets its own instance of the plugin class. The plugin factory PluginFactory class must be specialized to provide a factory for new objects of the Plugin class. See the documentation there for details about the plugin mechanism and the basic concepts.

This class has been introduced in version 0.22.

Public constructors

new Plugin ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[virtual]	void	**activated**		Gets called when the plugin is activated (selected in the tool bar)
[virtual]	void	**config_finalize**		Sends the post-configuration request to the plugin
[virtual]	bool	**configure**	(string name, string value)	Sends configuration requests to the plugin
[virtual]	void	**deactivated**		Gets called when the plugin is deactivated and another plugin is activated
[virtual]	void	**drag_cancel**		Gets called on various occasions when a drag operation should be canceled
[virtual]	bool	**enter_event**	(bool prio)	Handles the enter event (mouse enters canvas area of view)
	void	**grab_mouse**		Redirects mouse events to this plugin, even if the plugin is not active.

[virtual,const]	bool	**has_tracking_position**		Gets a value indicating whether the plugin provides a tracking position
[virtual]	bool	**key_event**	(unsigned int key, unsigned int buttons)	Handles the key pressed event
[virtual]	bool	**leave_event**	(bool prio)	Handles the leave event (mouse leaves canvas area of view)
[virtual]	void	**menu_activated**	(string symbol)	Gets called when a custom menu item is selected
[virtual]	bool	**mouse_button_pressed_event**	(DPoint p, unsigned int buttons, bool prio)	Handles the mouse button pressed event
[virtual]	bool	**mouse_button_released**	(DPoint p, unsigned int buttons, bool prio)	Handles the mouse button release event
[virtual]	bool	**mouse_click_event**	(DPoint p, unsigned int buttons, bool prio)	Handles the mouse button click event (after the button has been released)
[virtual]	bool	**mouse_double_click_ev**	(DPoint p, unsigned int buttons, bool prio)	Handles the mouse button double-click event
[virtual]	bool	**mouse_moved_event**	(DPoint p, unsigned int buttons, bool prio)	Handles the mouse move event
	void	**set_cursor**	(int cursor_type)	Sets the cursor in the view area to the given type
[virtual,const]	DPoint	**tracking_position**		Gets the tracking position
	void	**ungrab_mouse**		Removes a mouse grab registered with grab_mouse.
[virtual]	void	**update**		Gets called when the view has changed
[virtual]	bool	**wheel_event**	(int delta, bool horizontal, DPoint p, unsigned int buttons, bool prio)	

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead

	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

activated

Signature: *[virtual]* void **activated**

Description: Gets called when the plugin is activated (selected in the tool bar)

config_finalize

Signature: *[virtual]* void **config_finalize**

Description: Sends the post-configuration request to the plugin

After all configuration parameters have been sent, 'config_finalize' is called to given the plugin a chance to update its internal state according to the new configuration.

configure

Signature: *[virtual]* bool **configure** (string name, string value)

Description: Sends configuration requests to the plugin

name:	The name of the configuration variable as registered in the plugin factory
value:	The value of the configuration variable

When a configuration variable is changed, the new value is reported to the plugin by calling the 'configure' method.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

deactivated

Signature: *[virtual]* void **deactivated**

Description: Gets called when the plugin is deactivated and another plugin is activated

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

drag_cancel

Signature: *[virtual]* void **drag_cancel**

Description: Gets called on various occasions when a drag operation should be canceled

If the plugin implements some press-and-drag or a click-and-drag operation, this callback should cancel this operation and return in some state waiting for a new mouse event.

enter_event

Signature: *[virtual]* bool **enter_event** (bool prio)

Description: Handles the enter event (mouse enters canvas area of view)

The behaviour of this callback is the same than for mouse_press_event, except that it is called when the mouse enters the canvas area. This method does not have a position nor button flags.

grab_mouse

Signature: void **grab_mouse**

Description: Redirects mouse events to this plugin, even if the plugin is not active.

has_tracking_position

Signature: *[virtual,const]* bool **has_tracking_position**

Description: Gets a value indicating whether the plugin provides a tracking position

The tracking position is shown in the lower-left corner of the layout window to indicate the current position. If this method returns true for the active service, the application will fetch the position by calling tracking_position rather than displaying the original mouse position.

This method has been added in version 0.27.6.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

key_event

Signature: *[virtual]* bool **key_event** (unsigned int key, unsigned int buttons)

Description: Handles the key pressed event

key:	The Qt key code of the key that was pressed
buttons:	A combination of the constants in the ButtonState class which codes both the mouse buttons and the key modifiers (.e. ShiftButton etc).
Returns:	True to terminate dispatcher

This method will called by the view on the active plugin when a button is pressed on the mouse.

If the plugin handles the event, it should return true to indicate that the event should not be processed further.

leave_event

Signature: *[virtual]* bool **leave_event** (bool prio)

Description: Handles the leave event (mouse leaves canvas area of view)

The behaviour of this callback is the same than for mouse_press_event, except that it is called when the mouse leaves the canvas area. This method does not have a position nor button flags.

menu_activated

Signature: *[virtual]* void **menu_activated** (string symbol)

Description: Gets called when a custom menu item is selected

When a menu item is clicked which was registered with the plugin factory, the plugin's 'menu_activated' method is called for the current view. The symbol registered for the menu item is passed in the 'symbol' argument.

mouse_button_pressed_event

Signature: *[virtual]* bool **mouse_button_pressed_event** (DPoint p, unsigned int buttons, bool prio)

Description: Handles the mouse button pressed event

p:	The point at which the button was pressed

buttons: A combination of the constants in the ButtonState class which codes both the mouse buttons and the key modifiers (.e. LeftButton, ShiftButton etc).

Returns: True to terminate dispatcher

This method will called by the view when a button is pressed on the mouse.

First, the plugins that grabbed the mouse with grab_mouse will receive this event with 'prio' set to true in the reverse order the plugins grabbed the mouse. The loop will terminate if one of the mouse event handlers returns true.

If that is not the case or no plugin has grabbed the mouse, the active plugin receives the mouse event with 'prio' set to true.

If no receiver accepted the mouse event by returning true, it is sent again to all plugins with 'prio' set to false. Again, the loop terminates if one of the receivers returns true. The second pass gives inactive plugins a chance to monitor the mouse and implement specific actions - i.e. displaying the current position.

This event is not sent immediately when the mouse button is pressed but when a signification movement for the mouse cursor away from the original position is detected. If the mouse button is released before that, a mouse_clicked_event is sent rather than a press-move-release sequence.

mouse_button_released_ever

Signature: *[virtual]* bool **mouse_button_released_event** (DPoint p, unsigned int buttons, bool prio)

Description: Handles the mouse button release event

The behaviour of this callback is the same than for mouse_press_event, except that it is called when the mouse button is released.

mouse_click_event

Signature: *[virtual]* bool **mouse_click_event** (DPoint p, unsigned int buttons, bool prio)

Description: Handles the mouse button click event (after the button has been released)

The behaviour of this callback is the same than for mouse_press_event, except that it is called when the mouse button has been released without moving it.

mouse_double_click_event

Signature: *[virtual]* bool **mouse_double_click_event** (DPoint p, unsigned int buttons, bool prio)

Description: Handles the mouse button double-click event

The behaviour of this callback is the same than for mouse_press_event, except that it is called when the mouse button has been double-clicked.

mouse_moved_event

Signature: *[virtual]* bool **mouse_moved_event** (DPoint p, unsigned int buttons, bool prio)

Description: Handles the mouse move event

The behaviour of this callback is the same than for mouse_press_event, except that it is called when the mouse is moved in the canvas area.

new

Signature: *[static]* new Plugin ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

set_cursor

Signature: void **set_cursor** (int cursor_type)

Description: Sets the cursor in the view area to the given type

Setting the cursor has an effect only inside event handlers, i.e. mouse_press_event. The cursor is not set permanently. Is is reset in the mouse move handler unless a button is pressed or the cursor is explicitly set again in the mouse_move_event.

The cursor type is one of the cursor constants in the Cursor class, i.e. 'CursorArrow' for the normal cursor.

tracking_position

Signature: *[virtual,const]* DPoint **tracking_position**

Description: Gets the tracking position

See has_tracking_position for details.

This method has been added in version 0.27.6.

ungrab_mouse

Signature: void **ungrab_mouse**

Description: Removes a mouse grab registered with grab_mouse.

update

Signature: *[virtual]* void **update**

Description: Gets called when the view has changed

This method is called in particular if the view has changed the visible rectangle, i.e. after zooming in or out or panning. This callback can be used to update any internal states that depend on the view's state.

wheel_event

Signature: *[virtual]* bool **wheel_event** (int delta, bool horizontal, DPoint p, unsigned int buttons, bool prio)

Description:

The behaviour of this callback is the same than for mouse_press_event, except that it is called when the mouse wheel is rotated. Additional parameters for this event are 'delta' (the rotation angle in units of 1/8th degree) and 'horizontal' which is true when the horizontal wheel was rotated and false if the vertical wheel was rotated.

4.206. API reference - Class Cursor

Notation used in Ruby API documentation

Module: lay

Description: The namespace for the cursor constants

This class defines the constants for the cursor setting (for example for class Plugin, method set_cursor). This class has been introduced in version 0.22.

Public constructors

new Cursor ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const Cursor other)	Assigns another object to self
[const]	new Cursor ptr	**dup**		Creates a copy of self

Public static methods and constants

int	**Arrow**	'Arrow cursor' constant
int	**Blank**	'Blank cursor' constant
int	**Busy**	'Busy state cursor' constant
int	**ClosedHand**	'Closed hand cursor' constant
int	**Cross**	'Cross cursor' constant
int	**Forbidden**	'Forbidden area cursor' constant
int	**IBeam**	'I beam (text insert) cursor' constant
int	**None**	'No cursor (default)' constant for set_cursor (resets cursor to default)

int	OpenHand	'Open hand cursor' constant
int	PointingHand	'Pointing hand cursor' constant
int	SizeAll	'Size all directions cursor' constant
int	SizeBDiag	'Backward diagonal resize cursor' constant
int	SizeFDiag	'Forward diagonal resize cursor' constant
int	SizeHor	'Horizontal resize cursor' constant
int	SizeVer	'Vertical resize cursor' constant
int	SplitH	'split_horizontal cursor' constant
int	SplitV	'Split vertical cursor' constant
int	UpArrow	'Upward arrow cursor' constant
int	Wait	'Waiting cursor' constant
int	WhatsThis	'Question mark cursor' constant

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

Arrow

Signature: *[static]* int **Arrow**

Description: 'Arrow cursor' constant

Python specific notes:
The object exposes a readable attribute 'Arrow'. This is the getter.

Blank

Signature: *[static]* int **Blank**

Description: 'Blank cursor' constant

Python specific notes:
The object exposes a readable attribute 'Blank'. This is the getter.

Busy

Signature: *[static]* int **Busy**

Description: 'Busy state cursor' constant

Python specific notes:
The object exposes a readable attribute 'Busy'. This is the getter.

ClosedHand	**Signature**: *[static]* int **ClosedHand** **Description**: 'Closed hand cursor' constant **Python specific notes:** The object exposes a readable attribute 'ClosedHand'. This is the getter.
Cross	**Signature**: *[static]* int **Cross** **Description**: 'Cross cursor' constant **Python specific notes:** The object exposes a readable attribute 'Cross'. This is the getter.
Forbidden	**Signature**: *[static]* int **Forbidden** **Description**: 'Forbidden area cursor' constant **Python specific notes:** The object exposes a readable attribute 'Forbidden'. This is the getter.
IBeam	**Signature**: *[static]* int **IBeam** **Description**: 'I beam (text insert) cursor' constant **Python specific notes:** The object exposes a readable attribute 'IBeam'. This is the getter.
None	**Signature**: *[static]* int **None** **Description**: 'No cursor (default)' constant for set_cursor (resets cursor to default) **Python specific notes:** This member is available as 'None_' in Python. The object exposes a readable attribute 'None_'. This is the getter.
OpenHand	**Signature**: *[static]* int **OpenHand** **Description**: 'Open hand cursor' constant **Python specific notes:** The object exposes a readable attribute 'OpenHand'. This is the getter.
PointingHand	**Signature**: *[static]* int **PointingHand** **Description**: 'Pointing hand cursor' constant **Python specific notes:** The object exposes a readable attribute 'PointingHand'. This is the getter.
SizeAll	**Signature**: *[static]* int **SizeAll** **Description**: 'Size all directions cursor' constant **Python specific notes:** The object exposes a readable attribute 'SizeAll'. This is the getter.
SizeBDiag	**Signature**: *[static]* int **SizeBDiag** **Description**: 'Backward diagonal resize cursor' constant **Python specific notes:** The object exposes a readable attribute 'SizeBDiag'. This is the getter.

SizeFDiag	**Signature**: *[static]* int **SizeFDiag** **Description**: 'Forward diagonal resize cursor' constant **Python specific notes:** The object exposes a readable attribute 'SizeFDiag'. This is the getter.
SizeHor	**Signature**: *[static]* int **SizeHor** **Description**: 'Horizontal resize cursor' constant **Python specific notes:** The object exposes a readable attribute 'SizeHor'. This is the getter.
SizeVer	**Signature**: *[static]* int **SizeVer** **Description**: 'Vertical resize cursor' constant **Python specific notes:** The object exposes a readable attribute 'SizeVer'. This is the getter.
SplitH	**Signature**: *[static]* int **SplitH** **Description**: 'split_horizontal cursor' constant **Python specific notes:** The object exposes a readable attribute 'SplitH'. This is the getter.
SplitV	**Signature**: *[static]* int **SplitV** **Description**: 'Split vertical cursor' constant **Python specific notes:** The object exposes a readable attribute 'SplitV'. This is the getter.
UpArrow	**Signature**: *[static]* int **UpArrow** **Description**: 'Upward arrow cursor' constant **Python specific notes:** The object exposes a readable attribute 'UpArrow'. This is the getter.
Wait	**Signature**: *[static]* int **Wait** **Description**: 'Waiting cursor' constant **Python specific notes:** The object exposes a readable attribute 'Wait'. This is the getter.
WhatsThis	**Signature**: *[static]* int **WhatsThis** **Description**: 'Question mark cursor' constant **Python specific notes:** The object exposes a readable attribute 'WhatsThis'. This is the getter.
_create	**Signature**: void **_create** **Description**: Ensures the C++ object is created Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Cursor other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

	Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
dup	**Signature**: *[const]* new Cursor ptr **dup** **Description**: Creates a copy of self **Python specific notes:** This method also implements '__copy__' and '__deepcopy__'.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**Signature**: *[static]* new Cursor ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.

4.207. API reference - Class ButtonState

Notation used in Ruby API documentation

Module: lay

Description: The namespace for the button state flags in the mouse events of the Plugin class.

This class defines the constants for the button state. In the event handler, the button state is indicated by a bitwise combination of these constants. See Plugin for further details. This class has been introduced in version 0.22.

Public constructors

new ButtonState ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const ButtonState other)	Assigns another object to self
[const]	new ButtonState ptr	**dup**		Creates a copy of self

Public static methods and constants

int	**AltKey**	Indicates that the Alt key is pressed
int	**ControlKey**	Indicates that the Control key is pressed
int	**LeftButton**	Indicates that the left mouse button is pressed
int	**MidButton**	Indicates that the middle mouse button is pressed
int	**RightButton**	Indicates that the right mouse button is pressed
int	**ShiftKey**	Indicates that the Shift key is pressed

Deprecated methods (protected, public, static, non-static and constructors)

void	**create**	Use of this method is deprecated. Use _create instead

	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

AltKey

Signature: *[static]* int **AltKey**

Description: Indicates that the Alt key is pressed

This constant is combined with other constants within ButtonState

Python specific notes:
The object exposes a readable attribute 'AltKey'. This is the getter.

ControlKey

Signature: *[static]* int **ControlKey**

Description: Indicates that the Control key is pressed

This constant is combined with other constants within ButtonState

Python specific notes:
The object exposes a readable attribute 'ControlKey'. This is the getter.

LeftButton

Signature: *[static]* int **LeftButton**

Description: Indicates that the left mouse button is pressed

This constant is combined with other constants within ButtonState

Python specific notes:
The object exposes a readable attribute 'LeftButton'. This is the getter.

MidButton

Signature: *[static]* int **MidButton**

Description: Indicates that the middle mouse button is pressed

This constant is combined with other constants within ButtonState

Python specific notes:
The object exposes a readable attribute 'MidButton'. This is the getter.

RightButton

Signature: *[static]* int **RightButton**

Description: Indicates that the right mouse button is pressed

This constant is combined with other constants within ButtonState

Python specific notes:
The object exposes a readable attribute 'RightButton'. This is the getter.

ShiftKey

Signature: *[static]* int **ShiftKey**

Description: Indicates that the Shift key is pressed

This constant is combined with other constants within ButtonState

Python specific notes:
The object exposes a readable attribute 'ShiftKey'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const ButtonState other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new ButtonState ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new ButtonState ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

4.208. API reference - Class KeyCode

Notation used in Ruby API documentation

Module: lay

Description: The namespace for the some key codes.

This namespace defines some key codes understood by built-in LayoutView components. When compiling with Qt, these codes are compatible with Qt's key codes. The key codes are intended to be used when directly interfacing with LayoutView in non-Qt-based environments.

This class has been introduced in version 0.28.

Public constructors

new KeyCode ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const KeyCode other)	Assigns another object to self
[const]	new KeyCode ptr	**dup**		Creates a copy of self

Public static methods and constants

int	**Backspace**	Indicates the Backspace key
int	**Backtab**	Indicates the Backtab key
int	**Delete**	Indicates the Delete key
int	**Down**	Indicates the Down key
int	**End**	Indicates the End key
int	**Enter**	Indicates the Enter key
int	**Escape**	Indicates the Escape key
int	**Home**	Indicates the Home key

	int	**Insert**	Indicates the Insert key
	int	**Left**	Indicates the Left key
	int	**PageDown**	Indicates the PageDown key
	int	**PageUp**	Indicates the PageUp key
	int	**Return**	Indicates the Return key
	int	**Right**	Indicates the Right key
	int	**Tab**	Indicates the Tab key
	int	**Up**	Indicates the Up key

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

Backspace

Signature: *[static]* int **Backspace**

Description: Indicates the Backspace key

Python specific notes:
The object exposes a readable attribute 'Backspace'. This is the getter.

Backtab

Signature: *[static]* int **Backtab**

Description: Indicates the Backtab key

Python specific notes:
The object exposes a readable attribute 'Backtab'. This is the getter.

Delete

Signature: *[static]* int **Delete**

Description: Indicates the Delete key

Python specific notes:
The object exposes a readable attribute 'Delete'. This is the getter.

Down

Signature: *[static]* int **Down**

Description: Indicates the Down key

Python specific notes:
The object exposes a readable attribute 'Down'. This is the getter.

End

Signature: *[static]* int **End**

Description: Indicates the End key

Python specific notes:
The object exposes a readable attribute 'End'. This is the getter.

Enter

Signature: *[static]* int **Enter**

Description: Indicates the Enter key

Python specific notes:
The object exposes a readable attribute 'Enter'. This is the getter.

Escape

Signature: *[static]* int **Escape**

Description: Indicates the Escape key

Python specific notes:
The object exposes a readable attribute 'Escape'. This is the getter.

Home

Signature: *[static]* int **Home**

Description: Indicates the Home key

Python specific notes:
The object exposes a readable attribute 'Home'. This is the getter.

Insert

Signature: *[static]* int **Insert**

Description: Indicates the Insert key

Python specific notes:
The object exposes a readable attribute 'Insert'. This is the getter.

Left

Signature: *[static]* int **Left**

Description: Indicates the Left key

Python specific notes:
The object exposes a readable attribute 'Left'. This is the getter.

PageDown

Signature: *[static]* int **PageDown**

Description: Indicates the PageDown key

Python specific notes:
The object exposes a readable attribute 'PageDown'. This is the getter.

PageUp

Signature: *[static]* int **PageUp**

Description: Indicates the PageUp key

Python specific notes:
The object exposes a readable attribute 'PageUp'. This is the getter.

Return

Signature: *[static]* int **Return**

Description: Indicates the Return key

Python specific notes:
The object exposes a readable attribute 'Return'. This is the getter.

Right

Signature: *[static]* int **Right**

Description: Indicates the Right key

Python specific notes:
The object exposes a readable attribute 'Right'. This is the getter.

Tab

Signature: *[static]* int **Tab**

Description: Indicates the Tab key

Python specific notes:
The object exposes a readable attribute 'Tab'. This is the getter.

Up

Signature: *[static]* int **Up**

Description: Indicates the Up key

Python specific notes:
The object exposes a readable attribute 'Up'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const KeyCode other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new KeyCode ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new KeyCode ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

4.209. API reference - Class Dispatcher

Notation used in Ruby API documentation

Module: lay

Description: Root of the configuration space in the plugin context and menu dispatcher

This class provides access to the root configuration space in the context of plugin programming. You can use this class to obtain configuration parameters from the configuration tree during plugin initialization. However, the preferred way of plugin configuration is through Plugin#configure.

Currently, the application object provides an identical entry point for configuration modification. For example, "Application::instance.set_config" is identical to "Dispatcher::instance.set_config". Hence there is little motivation for the Dispatcher class currently and this interface may be modified or removed in the future. This class has been introduced in version 0.25 as 'PluginRoot'. It is renamed and enhanced as 'Dispatcher' in 0.27.

Public constructors

new Dispatcher ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**clear_config**		Clears the configuration parameters
	void	**commit_config**		Commits the configuration settings
	variant	**get_config**	(string name)	Gets the value of a local configuration parameter
	string[]	**get_config_names**		Gets the configuration parameter names
	bool	**read_config**	(string file_name)	Reads the configuration from a file
	void	**set_config**	(string name, string value)	Set a local configuration parameter with the given name to the given value
	bool	**write_config**	(string file_name)	Writes configuration to a file

Public static methods and constants

Dispatcher ptr	instance	Gets the singleton instance of the Dispatcher object

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void _**manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

clear_config

Signature: void **clear_config**

Description: Clears the configuration parameters

commit_config

Signature: void **commit_config**

Description: Commits the configuration settings

Some configuration options are queued for performance reasons and become active only after 'commit_config' has been called. After a sequence of set_config calls, this method should be called to activate the settings made by these calls.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

get_config

Signature: variant **get_config** (string name)

Description: Gets the value of a local configuration parameter

name:	The name of the configuration parameter whose value shall be obtained (a string)
Returns:	The value of the parameter or nil if there is no such parameter

get_config_names

Signature: string[] **get_config_names**

Description: Gets the configuration parameter names

Returns: A list of configuration parameter names

This method returns the names of all known configuration parameters. These names can be used to get and set configuration parameter values.

instance

Signature: *[static]* Dispatcher ptr **instance**

Description: Gets the singleton instance of the Dispatcher object

Returns: The instance

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new Dispatcher ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

read_config

Signature: bool **read_config** (string file_name)

Description: Reads the configuration from a file

Returns: A value indicating whether the operation was successful

This method silently does nothing, if the config file does not exist. If it does and an error occurred, the error message is printed on stderr. In both cases, false is returned.

set_config

Signature: void **set_config** (string name, string value)

Description: Set a local configuration parameter with the given name to the given value

name: The name of the configuration parameter to set

value: The value to which to set the configuration parameter

This method sets a configuration parameter with the given name to the given value. Values can only be strings. Numerical values have to be converted into strings first. Local configuration parameters override global configurations for this specific view. This allows for example to override global settings of background colors. Any local settings are not written to the configuration file.

write_config

Signature: bool **write_config** (string file_name)

Description: Writes configuration to a file

Returns: A value indicating whether the operation was successful

If the configuration file cannot be written, false is returned but no exception is thrown.

4.210. API reference - Class BrowserDialog

Notation used in Ruby API documentation

Module: lay

Description: A HTML display and browser dialog

Class hierarchy: BrowserDialog » QDialog » QWidget » QObject

The browser dialog displays HTML code in a browser panel. The HTML code is delivered through a separate object of class BrowserSource which acts as a "server" for a specific kind of URL scheme. Whenever the browser sees a URL starting with "int:" it will ask the connected BrowserSource object for the HTML code of that page using its 'get' method. The task of the BrowserSource object is to format the data requested in HTML and deliver it.

One use case for that class is the implementation of rich data browsers for structured information. In a simple scenario, the browser dialog can be instantiated with a static HTML page. In that case, only the content of that page is shown.

Here's a simple example:

```
html = "<html><body>Hello, world!</body></html>"
RBA::BrowserDialog::new(html).exec
```

And that is an example for the use case with a BrowserSource as the "server":

```
class MySource < RBA::BrowserSource
  def get(url)
    if (url =~ /b.html$/)
      return "<html><body>The second page</body></html>"
    else
      return "<html><body>The first page with a <a href='int:b.html'>link</a></body></html>"
    end
  end
end

source = MySource::new
RBA::BrowserDialog::new(source).exec
```

Public constructors

new BrowserDialog ptr	**new**	(BrowserSource ptr source)	Creates a HTML browser window with a BrowserSource as the source of HTML code
new BrowserDialog ptr	**new**	(string html)	Creates a HTML browser window with a static HTML content
new BrowserDialog ptr	**new**	(QWidget ptr parent, BrowserSource ptr source)	Creates a HTML browser window with a BrowserSource as the source of HTML code
new BrowserDialog ptr	**new**	(QWidget ptr parent, string html)	Creates a HTML browser window with a static HTML content

Public methods

void	**_create**	Ensures the C++ object is created
void	**_destroy**	Explicitly destroys the object

[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**caption=**	(string caption)	Sets the caption of the window
	int	**execute**		Executes the HTML browser dialog as a modal window
	void	**home=**	(string home_url)	Sets the browser's initial and current URL which is selected if the "home" location is chosen
	void	**label=**	(string label)	Sets the label text
	void	**load**	(string url)	Loads the given URL into the browser dialog
	void	**reload**		Reloads the current page
	void	**resize**	(int width, int height)	Sets the size of the dialog window
	void	**search**	(string search_item)	Issues a search request using the given search item and the search URL specified with set_search_url
	void	**set_search_url**	(string url, string query_item)	Enables the search field and specifies the search URL generated for a search
	void	**source=**	(BrowserSource ptr source)	Connects to a source object

Deprecated methods (protected, public, static, non-static and constructors)

[virtual]	void	**closed**		Use of this method is deprecated
	int	**exec**		Use of this method is deprecated. Use execute instead
	void	**set_caption**	(string caption)	Use of this method is deprecated. Use caption= instead
	void	**set_home**	(string home_url)	Use of this method is deprecated. Use home= instead
	void	**set_size**	(int width, int height)	Use of this method is deprecated. Use resize instead
	void	**set_source**	(BrowserSource ptr source)	Use of this method is deprecated. Use source= instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

caption=

Signature: void **caption=** (string caption)

Description: Sets the caption of the window

Python specific notes:
The object exposes a writable attribute 'caption'. This is the setter.

closed

Signature: *[virtual]* void **closed**

Description: Callback when the dialog is closed

Use of this method is deprecated

This callback can be reimplemented to implement cleanup functionality when the dialog is closed.

exec

Signature: int **exec**

Description: Executes the HTML browser dialog as a modal window

Use of this method is deprecated. Use execute instead

Python specific notes:
This attribute is available as 'exec_' in Python.

execute

Signature: int **execute**

Description: Executes the HTML browser dialog as a modal window

Python specific notes:
This attribute is available as 'exec_' in Python.

home=

Signature: void **home=** (string home_url)

Description: Sets the browser's initial and current URL which is selected if the "home" location is chosen

The home URL is the one shown initially and the one which is selected when the "home" button is pressed. The default location is "int:/index.html".

Python specific notes:
The object exposes a writable attribute 'home'. This is the setter.

label=

Signature: void **label=** (string label)

Description: Sets the label text

The label is shown left of the navigation buttons. By default, no label is specified.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'label'. This is the setter.

load

Signature: void **load** (string url)

Description: Loads the given URL into the browser dialog

Typically the URL has the "int:" scheme so the HTML code is taken from the BrowserSource object.

new

(1) Signature: *[static]* new BrowserDialog ptr **new** (BrowserSource ptr source)

Description: Creates a HTML browser window with a BrowserSource as the source of HTML code

This method has been introduced in version 0.23.

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new BrowserDialog ptr **new** (string html)

Description: Creates a HTML browser window with a static HTML content

This method has been introduced in version 0.23.

Python specific notes:

This method is the default initializer of the object.

(3) Signature: *[static]* new BrowserDialog ptr **new** (QWidget ptr parent, BrowserSource ptr source)

Description: Creates a HTML browser window with a BrowserSource as the source of HTML code

This method variant with a parent argument has been introduced in version 0.24.2.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new BrowserDialog ptr **new** (QWidget ptr parent, string html)

Description: Creates a HTML browser window with a static HTML content

This method variant with a parent argument has been introduced in version 0.24.2.

Python specific notes:
This method is the default initializer of the object.

reload

Signature: void **reload**

Description: Reloads the current page

resize

Signature: void **resize** (int width, int height)

Description: Sets the size of the dialog window

search

Signature: void **search** (string search_item)

Description: Issues a search request using the given search item and the search URL specified with set_search_url

See set_search_url for a description of the search mechanism.

set_caption

Signature: void **set_caption** (string caption)

Description: Sets the caption of the window

Use of this method is deprecated. Use caption= instead

Python specific notes:
The object exposes a writable attribute 'caption'. This is the setter.

set_home

Signature: void **set_home** (string home_url)

Description: Sets the browser's initial and current URL which is selected if the "home" location is chosen

Use of this method is deprecated. Use home= instead

The home URL is the one shown initially and the one which is selected when the "home" button is pressed. The default location is "int:/index.html".

Python specific notes:
The object exposes a writable attribute 'home'. This is the setter.

set_search_url

Signature: void **set_search_url** (string url, string query_item)

Description: Enables the search field and specifies the search URL generated for a search

If a search URL is set, the search box right to the navigation bar will be enabled. When a text is entered into the search box, the browser will navigate to an URL composed of the search URL, the search item and the search text, i.e. "myurl?item=search_text".

This method has been introduced in version 0.23.

set_size

Signature: void **set_size** (int width, int height)

Description: Sets the size of the dialog window

Use of this method is deprecated. Use resize instead

set_source

Signature: void **set_source** (BrowserSource ptr source)

Description: Connects to a source object

Use of this method is deprecated. Use source= instead

Setting the source should be the first thing done after the BrowserDialog object is created. It will not have any effect after the browser has loaded the first page. In particular, home= should be called after the source was set.

Python specific notes:
The object exposes a writable attribute 'source'. This is the setter.

source=

Signature: void **source=** (BrowserSource ptr source)

Description: Connects to a source object

Setting the source should be the first thing done after the BrowserDialog object is created. It will not have any effect after the browser has loaded the first page. In particular, home= should be called after the source was set.

Python specific notes:
The object exposes a writable attribute 'source'. This is the setter.

4.211. API reference - Class BrowserSource

Notation used in Ruby API documentation

Module: lay

Description: The BrowserDialog's source for "int" URL's

The source object basically acts as a "server" for special URL's using "int" as the scheme. Classes that want to implement such functionality must derive from BrowserSource and reimplement the get method. This method is supposed to deliver a HTML page for the given URL.

Alternatively to implementing this functionality, a source object may be instantiated using the constructor with a HTML code string. This will create a source object that simply displays the given string as the initial and only page.

Public constructors

new BrowserSource ptr	**new**	(string html)	Constructs a BrowserSource object with a default HTML string

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const BrowserSource other)	Assigns another object to self
[const]	new BrowserSource ptr	**dup**		Creates a copy of self
[virtual]	string	**get**	(string url)	Gets the HTML code for a given "int" URL.
	QImage	**get_image**	(string url)	Gets the image object for a specific URL
	string	**next_topic**	(string url)	Gets the next topic URL from a given URL
	string	**prev_topic**	(string url)	Gets the previous topic URL from a given URL

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead

	void	destroy		Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?		Use of this method is deprecated. Use _is_const_object? instead
[static]	new BrowserSource ptr	new_html	(string html)	Use of this method is deprecated. Use new instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void _**manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void _**unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the

reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const BrowserSource other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new BrowserSource ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

get

Signature: *[virtual]* string **get** (string url)

Description: Gets the HTML code for a given "int" URL.

If this method returns an empty string, the browser will not be set to a new location. This allows implementing any functionality behind such links. If the method returns a string, the content of this string is displayed in the HTML browser page.

get_image

Signature: QImage **get_image** (string url)

Description: Gets the image object for a specific URL

This method has been introduced in version 0.28.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

	This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**Signature**: *[static]* new BrowserSource ptr **new** (string html) **Description**: Constructs a BrowserSource object with a default HTML string The default HTML string is sent when no specific implementation is provided. **Python specific notes:** This method is the default initializer of the object.
new_html	**Signature**: *[static]* new BrowserSource ptr **new_html** (string html) **Description**: Constructs a BrowserSource object with a default HTML string Use of this method is deprecated. Use new instead The default HTML string is sent when no specific implementation is provided. **Python specific notes:** This method is the default initializer of the object.
next_topic	**Signature**: string **next_topic** (string url) **Description**: Gets the next topic URL from a given URL An empty string will be returned if no next topic is available. This method has been introduced in version 0.28.
prev_topic	**Signature**: string **prev_topic** (string url) **Description**: Gets the previous topic URL from a given URL An empty string will be returned if no previous topic is available. This method has been introduced in version 0.28.

4.212. API reference - Class BrowserPanel

Notation used in Ruby API documentation

Module: lay

Description: A HTML display and browser widget

Class hierarchy: BrowserPanel » QWidget » QObject

This widget provides the functionality of BrowserDialog within a widget. It can be embedded into other dialogs. For details about the use model of this class see BrowserDialog.

This class has been introduced in version 0.25.

Public constructors

new BrowserPanel ptr	**new**	(QWidget ptr parent, BrowserSource ptr source)	Creates a HTML browser widget with a BrowserSource as the source of HTML code
new BrowserPanel ptr	**new**	(QWidget ptr parent)	Creates a HTML browser widget

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**home=**	(string home_url)	Sets the browser widget's initial and current URL which is selected if the "home" location is chosen
	void	**label=**	(string label)	Sets the label text
	void	**load**	(string url)	Loads the given URL into the browser widget
	void	**reload**		Reloads the current page
	void	**search**	(string search_item)	Issues a search request using the given search item and the search URL specified with set_search_url
	void	**set_search_url**	(string url, string query_item)	Enables the search field and specifies the search URL generated for a search
	void	**source=**	(BrowserSource ptr source)	Connects to a source object

[const]	string	**url**	Gets the URL currently shown

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

home=

Signature: void **home=** (string home_url)

Description: Sets the browser widget's initial and current URL which is selected if the "home" location is chosen

The home URL is the one shown initially and the one which is selected when the "home" button is pressed. The default location is "int:/index.html".

Python specific notes:
The object exposes a writable attribute 'home'. This is the setter.

label=

Signature: void **label=** (string label)

Description: Sets the label text

The label is shown left of the navigation buttons. By default, no label is specified.

Python specific notes:
The object exposes a writable attribute 'label'. This is the setter.

load

Signature: void **load** (string url)

Description: Loads the given URL into the browser widget

Typically the URL has the "int:" scheme so the HTML code is taken from the BrowserSource object.

new

(1) Signature: *[static]* new BrowserPanel ptr **new** (QWidget ptr parent, BrowserSource ptr source)

Description: Creates a HTML browser widget with a BrowserSource as the source of HTML code

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new BrowserPanel ptr **new** (QWidget ptr parent)

Description: Creates a HTML browser widget

Python specific notes:
This method is the default initializer of the object.

reload

Signature: void **reload**

Description: Reloads the current page

search

Signature: void **search** (string search_item)

Description: Issues a search request using the given search item and the search URL specified with set_search_url

See search_url= for a description of the search mechanism.

set_search_url

Signature: void **set_search_url** (string url, string query_item)

Description: Enables the search field and specifies the search URL generated for a search

If a search URL is set, the search box right to the navigation bar will be enabled. When a text is entered into the search box, the browser will navigate to an URL composed of the search URL, the search item and the search text, i.e. "myurl?item=search_text".

source=

Signature: void **source=** (BrowserSource ptr source)

Description: Connects to a source object

Setting the source should be the first thing done after the BrowserDialog object is created. It will not have any effect after the browser has loaded the first page. In particular, home= should be called after the source was set.

Python specific notes:

The object exposes a writable attribute 'source'. This is the setter.

url

Signature: *[const]* string **url**

Description: Gets the URL currently shown

4.213. API reference - Class InputDialog

Notation used in Ruby API documentation

Module: lay

Description: Various methods to open a dialog requesting data entry

This class provides some basic dialogs to enter a single value. Values can be strings floating-point values, integer values or an item from a list. This functionality is provided through the static (class) methods ask_...

Here are some examples:

```
# get a double value between -10 and 10 (initial value is 0):
v = RBA::InputDialog::ask_double_ex("Dialog Title", "Enter the value here:", 0, -10, 10, 1)
# get an item from a list:
v = RBA::InputDialog::ask_item("Dialog Title", "Select one:", [ "item 1", "item 2", "item 3" ], 1)
```

All these examples return the "nil" value if "Cancel" is pressed.

If you have enabled the Qt binding, you can use QInputDialog directly.

Public constructors

new InputDialog ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const InputDialog other)	Assigns another object to self
[const]	new InputDialog ptr	**dup**		Creates a copy of self

Public static methods and constants

variant	**ask_double**	(string title, string label, double value, int digits)		Open an input dialog requesting a floating-point value

	variant	**ask_double_ex**	(string title, string label, double value, double min, double max, int digits)	Open an input dialog requesting a floating-point value with enhanced capabilities
	variant	**ask_int**	(string title, string label, int value)	Open an input dialog requesting an integer value
	variant	**ask_int_ex**	(string title, string label, int value, int min, int max, int step)	Open an input dialog requesting an integer value with enhanced capabilities
	variant	**ask_item**	(string title, string label, string[] items, int value)	Open an input dialog requesting an item from a list
	variant	**ask_string**	(string title, string label, string value)	Open an input dialog requesting a string
	variant	**ask_string_passwo**	(string title, string label, string value)	Open an input dialog requesting a string without showing the actual characters entered

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[static]	DoubleValue	**get_double**	(string title, string label, double value, int digits)	Use of this method is deprecated
[static]	DoubleValue	**get_double_ex**	(string title, string label, double value, double min, double max, int digits)	Use of this method is deprecated
[static]	IntValue	**get_int**	(string title, string label, int value)	Use of this method is deprecated

[static]	IntValue	**get_int_ex**	(string title, string label, int value, int min, int max, int step)	Use of this method is deprecated
[static]	StringValue	**get_item**	(string title, string label, string[] items, int value)	Use of this method is deprecated
[static]	StringValue	**get_string**	(string title, string label, string value)	Use of this method is deprecated
[static]	StringValue	**get_string_password**	(string title, string label, string value)	Use of this method is deprecated
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known

not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

ask_double

Signature: *[static]* variant **ask_double** (string title, string label, double value, int digits)

Description: Open an input dialog requesting a floating-point value

title:	The title to display for the dialog
label:	The label text to display for the dialog
value:	The initial value for the input field
digits:	The number of digits allowed
Returns:	The value entered if "Ok" was pressed or nil if "Cancel" was pressed

This method has been introduced in 0.22 and is somewhat easier to use than the get_.. equivalent.

ask_double_ex

Signature: *[static]* variant **ask_double_ex** (string title, string label, double value, double min, double max, int digits)

Description: Open an input dialog requesting a floating-point value with enhanced capabilities

title:	The title to display for the dialog
label:	The label text to display for the dialog
value:	The initial value for the input field
min:	The minimum value allowed
max:	The maximum value allowed
digits:	The number of digits allowed
Returns:	The value entered if "Ok" was pressed or nil if "Cancel" was pressed

This method has been introduced in 0.22 and is somewhat easier to use than the get_.. equivalent.

ask_int

Signature: *[static]* variant **ask_int** (string title, string label, int value)

Description: Open an input dialog requesting an integer value

title:	The title to display for the dialog
label:	The label text to display for the dialog
value:	The initial value for the input field
Returns:	The value entered if "Ok" was pressed or nil if "Cancel" was pressed

This method has been introduced in 0.22 and is somewhat easier to use than the get_.. equivalent.

ask_int_ex

Signature: *[static]* variant **ask_int_ex** (string title, string label, int value, int min, int max, int step)

Description: Open an input dialog requesting an integer value with enhanced capabilities

title:	The title to display for the dialog
label:	The label text to display for the dialog
value:	The initial value for the input field
min:	The minimum value allowed
max:	The maximum value allowed
step:	The step size for the spin buttons
Returns:	The value entered if "Ok" was pressed or nil if "Cancel" was pressed

This method has been introduced in 0.22 and is somewhat easier to use than the get_.. equivalent.

ask_item

Signature: *[static]* variant **ask_item** (string title, string label, string[] items, int value)

Description: Open an input dialog requesting an item from a list

title:	The title to display for the dialog
label:	The label text to display for the dialog
items:	The list of items to show in the selection element
selection:	The initial selection (index of the element selected initially)
Returns:	The string of the item selected if "Ok" was pressed or nil if "Cancel" was pressed

This method has been introduced in 0.22 and is somewhat easier to use than the get_.. equivalent.

ask_string

Signature: *[static]* variant **ask_string** (string title, string label, string value)

Description: Open an input dialog requesting a string

title:	The title to display for the dialog
label:	The label text to display for the dialog
value:	The initial value for the input field
Returns:	The string entered if "Ok" was pressed or nil if "Cancel" was pressed

This method has been introduced in 0.22 and is somewhat easier to use than the get_.. equivalent.

ask_string_password

Signature: *[static]* variant **ask_string_password** (string title, string label, string value)

Description: Open an input dialog requesting a string without showing the actual characters entered

title:	The title to display for the dialog
label:	The label text to display for the dialog
value:	The initial value for the input field
Returns:	The string entered if "Ok" was pressed or nil if "Cancel" was pressed

This method has been introduced in 0.22 and is somewhat easier to use than the get_.. equivalent.

assign

Signature: void **assign** (const InputDialog other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new InputDialog ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

get_double

Signature: *[static]* DoubleValue **get_double** (string title, string label, double value, int digits)

Description: Open an input dialog requesting a floating-point value

title:	The title to display for the dialog
label:	The label text to display for the dialog
value:	The initial value for the input field
digits:	The number of digits allowed
Returns:	A DoubleValue object with has_value? set to true, if "Ok" was pressed and the value given in its value attribute

Use of this method is deprecated

Starting from 0.22, this method is deprecated and it is recommended to use the ask_... equivalent.

get_double_ex

Signature: *[static]* DoubleValue **get_double_ex** (string title, string label, double value, double min, double max, int digits)

Description: Open an input dialog requesting a floating-point value with enhanced capabilities

title:	The title to display for the dialog
label:	The label text to display for the dialog
value:	The initial value for the input field
min:	The minimum value allowed
max:	The maximum value allowed
digits:	The number of digits allowed

Returns: A DoubleValue object with has_value? set to true, if "Ok" was pressed and the value given in its value attribute

Use of this method is deprecated

Starting from 0.22, this method is deprecated and it is recommended to use the ask_... equivalent.

get_int

Signature: *[static]* IntValue **get_int** (string title, string label, int value)

Description: Open an input dialog requesting an integer value

title: The title to display for the dialog

label: The label text to display for the dialog

value: The initial value for the input field

Returns: A IntValue object with has_value? set to true, if "Ok" was pressed and the value given in its value attribute

Use of this method is deprecated

Starting from 0.22, this method is deprecated and it is recommended to use the ask_... equivalent.

get_int_ex

Signature: *[static]* IntValue **get_int_ex** (string title, string label, int value, int min, int max, int step)

Description: Open an input dialog requesting an integer value with enhanced capabilities

title: The title to display for the dialog

label: The label text to display for the dialog

value: The initial value for the input field

min: The minimum value allowed

max: The maximum value allowed

step: The step size for the spin buttons

Returns: A IntValue object with has_value? set to true, if "Ok" was pressed and the value given in its value attribute

Use of this method is deprecated

Starting from 0.22, this method is deprecated and it is recommended to use the ask_... equivalent.

get_item

Signature: *[static]* StringValue **get_item** (string title, string label, string[] items, int value)

Description: Open an input dialog requesting an item from a list

title: The title to display for the dialog

label: The label text to display for the dialog

items: The list of items to show in the selection element

selection: The initial selection (index of the element selected initially)

Returns: A StringValue object with has_value? set to true, if "Ok" was pressed and the value given in its value attribute

Use of this method is deprecated

Starting from 0.22, this method is deprecated and it is recommended to use the ask_... equivalent.

get_string

Signature: *[static]* StringValue **get_string** (string title, string label, string value)

Description: Open an input dialog requesting a string

title: The title to display for the dialog

label: The label text to display for the dialog

value: The initial value for the input field

Returns: A StringValue object with has_value? set to true, if "Ok" was pressed and the value given in its value attribute

Use of this method is deprecated

Starting from 0.22, this method is deprecated and it is recommended to use the ask_... equivalent.

get_string_password

Signature: *[static]* StringValue **get_string_password** (string title, string label, string value)

Description: Open an input dialog requesting a string without showing the actual characters entered

title: The title to display for the dialog

label: The label text to display for the dialog

value: The initial value for the input field

Returns: A StringValue object with has_value? set to true, if "Ok" was pressed and the value given in its value attribute

Use of this method is deprecated

Starting from 0.22, this method is deprecated and it is recommended to use the ask_... equivalent.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new InputDialog ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

4.214. API reference - Class FileDialog

Notation used in Ruby API documentation

Module: lay

Description: Various methods to request a file name

This class provides some basic dialogs to select a file or directory. This functionality is provided through the static (class) methods ask_...

Here are some examples:

```
# get an existing directory:
v = RBA::FileDialog::ask_existing_dir("Dialog Title", ".")
# get multiple files:
v = RBA::FileDialog::ask_open_file_names("Title", ".", "All files (*)")
# ask for one file name to save a file:
v = RBA::FileDialog::ask_save_file_name("Title", ".", "All files (*)")
```

All these examples return the "nil" value if "Cancel" is pressed.

If you have enabled the Qt binding, you can use QFileDialog directly.

Public constructors

new FileDialog ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const FileDialog other)	Assigns another object to self
[const]	new FileDialog ptr	**dup**		Creates a copy of self

Public static methods and constants

	variant	**ask_existing_dir**	(string title, string dir)	Open a dialog to select a directory
	variant	**ask_open_file_name**	(string title, string dir,	Select one file for opening

			string filter)	
	variant	ask_open_file_names	(string title, string dir, string filter)	Select one or multiple files for opening
	variant	ask_save_file_name	(string title, string dir, string filter)	Select one file for writing
	variant	ask_save_file_name_with_fi	(string title, string dir, string filter)	Select one file for writing

Deprecated methods (protected, public, static, non-static and constructors)

	void	create		Use of this method is deprecated. Use _create instead
	void	destroy		Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?		Use of this method is deprecated. Use _destroyed? instead
[static]	StringValue	get_existing_dir	(string title, string dir)	Use of this method is deprecated
[static]	StringValue	get_open_file_name	(string title, string dir, string filter)	Use of this method is deprecated
[static]	StringListValue	get_open_file_names	(string title, string dir, string filter)	Use of this method is deprecated
[static]	StringValue	get_save_file_name	(string title, string dir, string filter)	Use of this method is deprecated
[const]	bool	is_const_object?		Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

ask_existing_dir

Signature: *[static]* variant **ask_existing_dir** (string title, string dir)

Description: Open a dialog to select a directory

title:	The title of the dialog
dir:	The directory selected initially
Returns:	The directory path selected or "nil" if "Cancel" was pressed

This method has been introduced in version 0.23. It is somewhat easier to use than the get_... equivalent.

ask_open_file_name

Signature: *[static]* variant **ask_open_file_name** (string title, string dir, string filter)

Description: Select one file for opening

title:	The title of the dialog
dir:	The directory selected initially
filter:	The filters available, for example "Images (*.png *.xpm *.jpg);;Text files (*.txt);;XML files (*.xml)"
Returns:	The path of the file selected or "nil" if "Cancel" was pressed

This method has been introduced in version 0.23. It is somewhat easier to use than the get_... equivalent.

ask_open_file_names

Signature: *[static]* variant **ask_open_file_names** (string title, string dir, string filter)

Description: Select one or multiple files for opening

title:	The title of the dialog
dir:	The directory selected initially
filter:	The filters available, for example "Images (*.png *.xpm *.jpg);;Text files (*.txt);;XML files (*.xml)"
Returns:	An array with the file paths selected or "nil" if "Cancel" was pressed

This method has been introduced in version 0.23. It is somewhat easier to use than the get_... equivalent.

ask_save_file_name

Signature: *[static]* variant **ask_save_file_name** (string title, string dir, string filter)

Description: Select one file for writing

title:	The title of the dialog
dir:	The directory selected initially
filter:	The filters available, for example "Images (*.png *.xpm *.jpg);;Text files (*.txt);;XML files (*.xml)"
Returns:	The path of the file chosen or "nil" if "Cancel" was pressed

This method has been introduced in version 0.23. It is somewhat easier to use than the get_... equivalent.

ask_save_file_name_with_fi

Signature: *[static]* variant **ask_save_file_name_with_filter** (string title, string dir, string filter)

Description: Select one file for writing

title:	The title of the dialog
dir:	The directory selected initially
filter:	The filters available, for example "Images (*.png *.xpm *.jpg);;Text files (*.txt);;XML files (*.xml)"
Returns:	"nil" if "Cancel" was pressed, otherwise a pair: The path of the file chosen and the index selected file type (-1 if no specific type was selected)

This method has been introduced in version 0.28.11.

assign

Signature: void **assign** (const FileDialog other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new FileDialog ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

get_existing_dir

Signature: *[static]* StringValue **get_existing_dir** (string title, string dir)

Description: Open a dialog to select a directory

title:	The title of the dialog
dir:	The directory selected initially
Returns:	A StringValue object that contains the directory path selected or with has_value? = false if "Cancel" was pressed

Use of this method is deprecated

Starting with version 0.23 this method is deprecated. Use ask_existing_dir instead.

get_open_file_name

Signature: *[static]* StringValue **get_open_file_name** (string title, string dir, string filter)

Description: Select one file for opening

title:	The title of the dialog
dir:	The directory selected initially
filter:	The filters available, for example "Images (*.png *.xpm *.jpg);;Text files (*.txt);;XML files (*.xml)"
Returns:	A StringValue object that contains the files selected or with has_value? = false if "Cancel" was pressed

Use of this method is deprecated

Starting with version 0.23 this method is deprecated. Use ask_open_file_name instead.

get_open_file_names

Signature: *[static]* StringListValue **get_open_file_names** (string title, string dir, string filter)

Description: Select one or multiple files for opening

title:	The title of the dialog
dir:	The directory selected initially
filter:	The filters available, for example "Images (*.png *.xpm *.jpg);;Text files (*.txt);;XML files (*.xml)"

Returns: A StringListValue object that contains the files selected or with has_value? = false if "Cancel" was pressed

Use of this method is deprecated

Starting with version 0.23 this method is deprecated. Use ask_open_file_names instead.

get_save_file_name

Signature: *[static]* StringValue **get_save_file_name** (string title, string dir, string filter)

Description: Select one file for writing

title: The title of the dialog

dir: The directory selected initially

filter: The filters available, for example "Images (*.png *.xpm *.jpg);;Text files (*.txt);;XML files (*.xml)"

Returns: A StringValue object that contains the files selected or with has_value? = false if "Cancel" was pressed

Use of this method is deprecated

Starting with version 0.23 this method is deprecated. Use ask_save_file_name instead.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new FileDialog ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

4.215. API reference - Class MessageBox

Notation used in Ruby API documentation

Module: lay

Description: Various methods to display message boxes

Class hierarchy: MessageBox » QMainWindow » QWidget » QObject

This class provides some basic message boxes. This functionality is provided through the static (class) methods warning, question and so on.

Here is some example:

```
# issue a warning and ask whether to continue:
v = RBA::MessageBox::warning("Dialog Title", "Something happened. Continue?", RBA::MessageBox::Yes +
 RBA::MessageBox::No)
if v == RBA::MessageBox::Yes
  ... continue ...
end
```

If you have enabled the Qt binding, you can use QMessageBox directly.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const MessageE other)	Assigns another object to self
[const]	new MessageBox ptr	**dup**		Creates a copy of self

Public static methods and constants

int	**Abort**	A constant describing the 'Abort' button
int	**Cancel**	A constant describing the 'Cancel' button
int	**Ignore**	A constant describing the 'Ignore' button
int	**No**	A constant describing the 'No' button
int	**Ok**	A constant describing the 'Ok' button

int	**Retry**		A constant describing the 'Retry' button
int	**Yes**		A constant describing the 'Yes' button
int	**critical**	(string title, string text, int buttons)	Open a critical (error) message box
int	**info**	(string title, string text, int buttons)	Open a information message box
int	**question**	(string title, string text, int buttons)	Open a question message box
int	**warning**	(string title, string text, int buttons)	Open a warning message box

Deprecated methods (protected, public, static, non-static and constructors)

[static]	int	**b_abort**	Use of this method is deprecated. Use Abort instead
[static]	int	**b_cancel**	Use of this method is deprecated. Use Cancel instead
[static]	int	**b_ignore**	Use of this method is deprecated. Use Ignore instead
[static]	int	**b_no**	Use of this method is deprecated. Use No instead
[static]	int	**b_ok**	Use of this method is deprecated. Use Ok instead
[static]	int	**b_retry**	Use of this method is deprecated. Use Retry instead
[static]	int	**b_yes**	Use of this method is deprecated. Use Yes instead

Detailed description

Abort

Signature: *[static]* int **Abort**

Description: A constant describing the 'Abort' button

Python specific notes:
The object exposes a readable attribute 'Abort'. This is the getter.

Cancel

Signature: *[static]* int **Cancel**

Description: A constant describing the 'Cancel' button

Python specific notes:
The object exposes a readable attribute 'Cancel'. This is the getter.

Ignore

Signature: *[static]* int **Ignore**

Description: A constant describing the 'Ignore' button

Python specific notes:
The object exposes a readable attribute 'Ignore'. This is the getter.

No

Signature: *[static]* int **No**

Description: A constant describing the 'No' button

Python specific notes:
The object exposes a readable attribute 'No'. This is the getter.

Ok

Signature: *[static]* int **Ok**

Description: A constant describing the 'Ok' button

Python specific notes:
The object exposes a readable attribute 'Ok'. This is the getter.

Retry

Signature: *[static]* int **Retry**

Description: A constant describing the 'Retry' button

Python specific notes:
The object exposes a readable attribute 'Retry'. This is the getter.

Yes

Signature: *[static]* int **Yes**

Description: A constant describing the 'Yes' button

Python specific notes:
The object exposes a readable attribute 'Yes'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known

not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const MessageBox other)

Description: Assigns another object to self

b_abort

Signature: *[static]* int **b_abort**

Description: A constant describing the 'Abort' button

Use of this method is deprecated. Use Abort instead

Python specific notes:
The object exposes a readable attribute 'Abort'. This is the getter.

b_cancel

Signature: *[static]* int **b_cancel**

Description: A constant describing the 'Cancel' button

Use of this method is deprecated. Use Cancel instead

Python specific notes:
The object exposes a readable attribute 'Cancel'. This is the getter.

b_ignore

Signature: *[static]* int **b_ignore**

Description: A constant describing the 'Ignore' button

Use of this method is deprecated. Use Ignore instead

Python specific notes:
The object exposes a readable attribute 'Ignore'. This is the getter.

b_no

Signature: *[static]* int **b_no**

Description: A constant describing the 'No' button

Use of this method is deprecated. Use No instead

Python specific notes:
The object exposes a readable attribute 'No'. This is the getter.

b_ok

Signature: *[static]* int **b_ok**

Description: A constant describing the 'Ok' button

Use of this method is deprecated. Use Ok instead

Python specific notes:
The object exposes a readable attribute 'Ok'. This is the getter.

b_retry

Signature: *[static]* int **b_retry**

Description: A constant describing the 'Retry' button

Use of this method is deprecated. Use Retry instead

Python specific notes:
The object exposes a readable attribute 'Retry'. This is the getter.

b_yes

Signature: *[static]* int **b_yes**

Description: A constant describing the 'Yes' button

Use of this method is deprecated. Use Yes instead

Python specific notes:
The object exposes a readable attribute 'Yes'. This is the getter.

critical

Signature: *[static]* int **critical** (string title, string text, int buttons)

Description: Open a critical (error) message box

title:	The title of the window
text:	The text to show
buttons:	A combination (+) of button constants (Ok and so on) describing the buttons to show for the message box
Returns:	The button constant describing the button that was pressed

dup

Signature: *[const]* new MessageBox ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

info

Signature: *[static]* int **info** (string title, string text, int buttons)

Description: Open a information message box

title:	The title of the window
text:	The text to show
buttons:	A combination (+) of button constants (Ok and so on) describing the buttons to show for the message box
Returns:	The button constant describing the button that was pressed

question

Signature: *[static]* int **question** (string title, string text, int buttons)

Description: Open a question message box

title:	The title of the window
text:	The text to show
buttons:	A combination (+) of button constants (Ok and so on) describing the buttons to show for the message box
Returns:	The button constant describing the button that was pressed

warning

Signature: *[static]* int **warning** (string title, string text, int buttons)

Description: Open a warning message box

title:	The title of the window

text:	The text to show
buttons:	A combination (+) of button constants (Ok and so on) describing the buttons to show for the message box
Returns:	The button constant describing the button that was pressed

4.216. API reference - Class NetlistObjectPath

Notation used in Ruby API documentation

Module: lay

Description: An object describing the instantiation of a netlist object.

This class describes the instantiation of a net or a device or a circuit in terms of a root circuit and a subcircuit chain leading to the indicated object.

See net= or device= for the indicated object, path= for the subcircuit chain.

This class has been introduced in version 0.27.

Public constructors

new NetlistObjectPath ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetlistObjectPa other)	Assigns another object to self
[const]	Device ptr	**device**		Gets the device the path points to.
	void	**device=**	(Device ptr device)	Sets the device the path points to.
[const]	new NetlistObjectPath ptr	**dup**		Creates a copy of self
[const]	bool	**is_null?**		Returns a value indicating whether the path is an empty one.
[const]	Net ptr	**net**		Gets the net the path points to.
	void	**net=**	(Net ptr net)	Sets the net the path points to.
[const]	SubCircuit ptr[]	**path**		Gets the path.

	void	path=	(SubCircuit ptr[] path)	Sets the path.
[const]	Circuit ptr	root		Gets the root circuit of the path.
	void	root=	(Circuit ptr root)	Sets the root circuit of the path.

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void _**manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

	Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
assign	**Signature**: void **assign** (const NetlistObjectPath other) **Description**: Assigns another object to self
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.
destroyed?	**Signature**: *[const]* bool **destroyed?** **Description**: Returns a value indicating whether the object was already destroyed Use of this method is deprecated. Use _destroyed? instead This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
device	**Signature**: *[const]* Device ptr **device** **Description**: Gets the device the path points to. **Python specific notes:** The object exposes a readable attribute 'device'. This is the getter.
device=	**Signature**: void **device=** (Device ptr device) **Description**: Sets the device the path points to. If the path describes the location of a device, this member will indicate it. The other way to describe a final object is net=. If neither a device nor net is given, the path describes a circuit and how it is referenced from the root. **Python specific notes:** The object exposes a writable attribute 'device'. This is the setter.

dup

Signature: *[const]* new NetlistObjectPath ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_null?

Signature: *[const]* bool **is_null?**

Description: Returns a value indicating whether the path is an empty one.

net

Signature: *[const]* Net ptr **net**

Description: Gets the net the path points to.

Python specific notes:
The object exposes a readable attribute 'net'. This is the getter.

net=

Signature: void **net=** (Net ptr net)

Description: Sets the net the path points to.

If the path describes the location of a net, this member will indicate it. The other way to describe a final object is device=. If neither a device nor net is given, the path describes a circuit and how it is referenced from the root.

Python specific notes:
The object exposes a writable attribute 'net'. This is the setter.

new

Signature: *[static]* new NetlistObjectPath ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

path

Signature: *[const]* SubCircuit ptr[] **path**

Description: Gets the path.

Python specific notes:
The object exposes a readable attribute 'path'. This is the getter.

path=

Signature: void **path=** (SubCircuit ptr[] path)

Description: Sets the path.

The path is a list of subcircuits leading from the root to the final object. The final (net, device) object is located in the circuit called by the last subcircuit of the subcircuit chain. If the subcircuit list is empty, the final object is located inside the root object.

Python specific notes:
The object exposes a writable attribute 'path'. This is the setter.

root

Signature: *[const]* Circuit ptr **root**

Description: Gets the root circuit of the path.

Python specific notes:
The object exposes a readable attribute 'root'. This is the getter.

root=

Signature: void **root=** (Circuit ptr root)

Description: Sets the root circuit of the path.

The root circuit is the circuit from which the path starts.

Python specific notes:
The object exposes a writable attribute 'root'. This is the setter.

4.217. API reference - Class NetlistObjectsPath

Notation used in Ruby API documentation

Module: lay

Description: An object describing the instantiation of a single netlist object or a pair of those.

This class is basically a pair of netlist object paths (see NetlistObjectPath). When derived from a single netlist view, only the first path is valid and will point to the selected object (a net, a device or a circuit). The second path is null.

If the path is derived from a paired netlist view (a LVS report view), the first path corresponds to the object in the layout netlist, the second one to the object in the schematic netlist. If the selected object isn't a matched one, either the first or second path may be a null or a partial path without a final net or device object or a partial path.

This class has been introduced in version 0.27.

Public constructors

new NetlistObjectsPath ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetlistObj other)	Assigns another object to self
[const]	new NetlistObjectsPath ptr	**dup**		Creates a copy of self
[const]	NetlistObjectPath	**first**		Gets the first object's path.
[const]	NetlistObjectPath	**second**		Gets the second object's path.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead

[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const NetlistObjectsPath other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetlistObjectsPath ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

first

Signature: *[const]* NetlistObjectPath **first**

Description: Gets the first object's path.

In cases of paired netlists (LVS database), the first path points to the layout netlist object. For the single netlist, the first path is the only path supplied.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new NetlistObjectsPath ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

second

Signature: *[const]* NetlistObjectPath **second**

Description: Gets the second object's path.

In cases of paired netlists (LVS database), the first path points to the schematic netlist object. For the single netlist, the second path is always a null path.

4.218. API reference - Class NetlistBrowserDialog

Notation used in Ruby API documentation

Module: lay

Description: Represents the netlist browser dialog.

This dialog is a part of the LayoutView class and can be obtained through LayoutView#netlist_browser. This interface allows to interact with the browser - mainly to get information about state changes.

This class has been introduced in version 0.27.

Public constructors

new NetlistBrowserDialog ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	NetlistObjectsPath	**current_path**		Gets the path of the current object as a path pair (combines layout and schematic object paths in case of a LVS database view).
	NetlistObjectPath	**current_path_first**		Gets the path of the current object on the first (layout in case of LVS database) side.
	NetlistObjectPath	**current_path_second**		Gets the path of the current object on the second (schematic in case of LVS database) side.
	LayoutToNetlist ptr	**db**		Gets the database the browser is connected to.
[signal]	void	**on_current_db_changed**		This event is triggered when the current database is changed.
[signal]	void	**on_probe**	(NetlistObjectPath first_path, NetlistObjectPath second_path)	This event is triggered when a net is probed.
[signal]	void	**on_selection_changed**		This event is triggered when the selection changed.
[const]	NetlistObjectsPath[]	**selected_paths**		Gets the nets currently selected objects (paths) in the netlist database browser.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

current_path

Signature: *[const]* NetlistObjectsPath **current_path**

Description: Gets the path of the current object as a path pair (combines layout and schematic object paths in case of a LVS database view).

current_path_first

Signature: NetlistObjectPath **current_path_first**

Description: Gets the path of the current object on the first (layout in case of LVS database) side.

current_path_second

Signature: NetlistObjectPath **current_path_second**

Description: Gets the path of the current object on the second (schematic in case of LVS database) side.

db

Signature: LayoutToNetlist ptr **db**

Description: Gets the database the browser is connected to.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new NetlistBrowserDialog ptr **new**

Description: Creates a new object of this class

	Python specific notes: This method is the default initializer of the object.
on_current_db_changed	**Signature**: *[signal]* void **on_current_db_changed** **Description**: This event is triggered when the current database is changed. The current database can be obtained with db. **Python specific notes:** The object exposes a readable attribute 'on_current_db_changed'. This is the getter. The object exposes a writable attribute 'on_current_db_changed'. This is the setter.
on_probe	**Signature**: *[signal]* void **on_probe** (NetlistObjectPath first_path, NetlistObjectPath second_path) **Description**: This event is triggered when a net is probed. The first path will indicate the location of the probed net in terms of two paths: one describing the instantiation of the net in layout space and one in schematic space. Both objects are NetlistObjectPath objects which hold the root circuit, the chain of subcircuits leading to the circuit containing the net and the net itself. **Python specific notes:** The object exposes a readable attribute 'on_probe'. This is the getter. The object exposes a writable attribute 'on_probe'. This is the setter.
on_selection_changed	**Signature**: *[signal]* void **on_selection_changed** **Description**: This event is triggered when the selection changed. The selection can be obtained with current_path_first, current_path_second, selected_nets, selected_devices, selected_subcircuits and selected_circuits. **Python specific notes:** The object exposes a readable attribute 'on_selection_changed'. This is the getter. The object exposes a writable attribute 'on_selection_changed'. This is the setter.
selected_paths	**Signature**: *[const]* NetlistObjectsPath[] **selected_paths** **Description**: Gets the nets currently selected objects (paths) in the netlist database browser. The result is an array of path pairs. See NetlistObjectsPath for details about these pairs.

4.219. API reference - Class LayoutViewWidget

Notation used in Ruby API documentation

Module: lay

Description:

Class hierarchy: LayoutViewWidget » QFrame » QWidget » QObject

This object produces a widget which embeds a LayoutView. This widget can be used inside Qt widget hierarchies. To access the LayoutView object within, use view.

This class has been introduced in version 0.28.

Public constructors

new LayoutViewWidget ptr	**new**	(QWidget ptr parent, bool editable = false, Manager ptr manager = nil, unsigned int options = 0)	Creates a standalone view widget

Public methods

	void	**_create**	Ensures the C++ object is created
	void	**_destroy**	Explicitly destroys the object
[const]	bool	**_destroyed?**	Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**	Returns a value indicating whether the reference is a const reference
	void	**_manage**	Marks the object as managed by the script side.
	void	**_unmanage**	Marks the object as no longer owned by the script side.
	QWidget ptr	**bookmarks_frame**	Gets the bookmarks side widget
	QWidget ptr	**hierarchy_control_frame**	Gets the cell view (hierarchy view) side widget
	QWidget ptr	**layer_control_frame**	Gets the layer control side widget
	QWidget ptr	**layer_toolbox_frame**	Gets the layer toolbox side widget
	QWidget ptr	**libraries_frame**	Gets the library view side widget
	LayoutView ptr	**view**	Gets the embedded view object.

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

bookmarks_frame

Signature: QWidget ptr **bookmarks_frame**

Description: Gets the bookmarks side widget

For details about side widgets see layer_control_frame.

This method has been introduced in version 0.27

hierarchy_control_frame

Signature: QWidget ptr **hierarchy_control_frame**

Description: Gets the cell view (hierarchy view) side widget

For details about side widgets see layer_control_frame.

This method has been introduced in version 0.27

layer_control_frame

Signature: QWidget ptr **layer_control_frame**

Description: Gets the layer control side widget

A 'side widget' is a widget attached to the view. It does not have a parent, so you can embed it into a different context. Please note that with embedding through 'setParent' it will be destroyed when your parent widget gets destroyed. It will be lost then to the view.

The side widget can be configured through the views configuration interface.

This method has been introduced in version 0.27

layer_toolbox_frame

Signature: QWidget ptr **layer_toolbox_frame**

Description: Gets the layer toolbox side widget

A 'side widget' is a widget attached to the view. It does not have a parent, so you can embed it into a different context. Please note that with embedding through 'setParent' it will be destroyed when your parent widget gets destroyed. It will be lost then to the view.

The side widget can be configured through the views configuration interface.

This method has been introduced in version 0.28

libraries_frame

Signature: QWidget ptr **libraries_frame**

Description: Gets the library view side widget

For details about side widgets see layer_control_frame.

This method has been introduced in version 0.27

new

Signature: *[static]* new LayoutViewWidget ptr **new** (QWidget ptr parent, bool editable = false, Manager ptr manager = nil, unsigned int options = 0)

Description: Creates a standalone view widget

parent:	The parent widget in which to embed the view
editable:	True to make the view editable
manager:	The Manager object to enable undo/redo
options:	A combination of the values in the LV_... constants from LayoutViewBase

This constructor has been introduced in version 0.25. It has been enhanced with the arguments in version 0.27.

Python specific notes:
This method is the default initializer of the object.

view

Signature: LayoutView ptr **view**

Description: Gets the embedded view object.

4.220. API reference - Class LayoutView

Notation used in Ruby API documentation

Module: lay

Description: The view object presenting one or more layout objects

Class hierarchy: LayoutView

Sub-classes: SelectionMode

The visual part of the view is the tab panel in the main window. The non-visual part are the redraw thread, the layout handles, cell lists, layer view lists etc. This object controls these aspects of the view and controls the appearance of the data.

Public constructors

new LayoutView ptr	**new**	(bool editable = false, Manager ptr manager = nil, unsigned int options = 0)	Creates a standalone view

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	CellView	**active_cellview**		Gets the active cellview (shown in hierarchy browser)
[const]	int	**active_cellview_index**		Gets the index of the active cellview (shown in hierarchy browser)
	void	**active_setview_index=**	(int index)	Makes the cellview with the given index the active one (shown in hierarchy browser)
	unsigned int	**add_l2ndb**	(LayoutToNetlist ptr db)	Adds the given netlist database to the view
	unsigned int	**add_line_style**	(string name, unsigned int data, unsigned int bits)	Adds a custom line style
	unsigned int	**add_line_style**	(string name, string string)	Adds a custom line style from a string

	unsigned int	**add_lvsdb**	(LayoutVsSchematic ptr db)	Adds the given database to the view
	void	**add_missing_layers**		Adds new layers to layer list
	unsigned int	**add_rdb**	(ReportDatabase ptr db)	Adds the given report database to the view
	unsigned int	**add_stipple**	(string name, unsigned int[] data, unsigned int bits)	Adds a stipple pattern
	unsigned int	**add_stipple**	(string name, string string)	Adds a stipple pattern given by a string
	Annotation	**annotation**	(int id)	Gets the annotation given by an ID
	variant[][]	**annotation_templates**		Gets a list of Annotation objects representing the annotation templates.
	InstElement	**ascend**	(int index)	Ascends upwards in the hierarchy.
[const]	LayerPropertiesIterat	**begin_layers**		Begin iterator for the layers
[const]	LayerPropertiesIterator	**begin_layers**	(unsigned int index)	Begin iterator for the layers
	void	**bookmark_view**	(string name)	Bookmarks the current view under the given name
[const]	DBox	**box**		Returns the displayed box in micron space
	void	**call_menu**	(string symbol)	Calls the menu item with the provided symbol.
	void	**cancel**		Cancels all edit operations
	CellView	**cellview**	(unsigned int cv_index)	Gets the cellview object for a given index
[const]	unsigned int	**cellviews**		Gets the number of cellviews
	void	**clear_annotations**		Clears all annotations on this view
	void	**clear_config**		Clears the local configuration parameters
	void	**clear_images**		Clear all images on this view
	void	**clear_layers**		Clears all layers
	void	**clear_layers**	(unsigned int index)	Clears all layers for the given layer properties list
	void	**clear_line_styles**		Removes all custom line styles
[const]	void	**clear_object_selection**		Clears the selection of geometrical objects (shapes or cell instances)

	void	clear_selection		Clears the selection of all objects (shapes, annotations, images ...)
	void	clear_stipples		Removes all custom line styles
	void	clear_transactions		Clears all transactions
	void	clear_transient_selection		Clears the transient selection (mouse-over hightlights) of all objects (shapes, annotations, images ...)
	void	close		Closes the view
	void	commit		Ends a transaction
	void	commit_config		Commits the configuration settings
	unsigned int	create_l2ndb	(string name)	Creates a new netlist database and returns the index of the new database
	unsigned int	create_layout	(bool add_cellview)	Creates a new, empty layout
	unsigned int	create_layout	(string tech, bool add_cellview)	Create a new, empty layout and associate it with the given technology
	unsigned int	create_layout	(string tech, bool add_cellview, bool init_layers)	Create a new, empty layout and associate it with the given technology
	unsigned int	create_lvsdb	(string name)	Creates a new netlist database and returns the index of the new database
	Annotation	create_measure_ruler	(const DPoint point, int ac = Annotation#AngleAny)	Createas an auto-measure ruler at the given point.
	unsigned int	create_rdb	(string name)	Creates a new report database and returns the index of the new database
[const]	LayerPropertiesIterator	current_layer		Gets the current layer view
	void	current_layer=	(const LayerPropertiesIterator iter)	Sets the current layer view
[const]	unsigned int	current_layer_list		Gets the index of the currently selected layer properties tab
	void	current_layer_list=	(unsigned int index)	Sets the index of the currently selected layer properties tab
	void	delete_layer	(LayerPropertiesIterator iter)	Deletes the layer properties node specified by the iterator
	void	delete_layer	(unsigned int index, LayerPropertiesIterator iter)	Deletes the layer properties node specified by the iterator
	void	delete_layer_list	(unsigned int index)	Deletes the given properties list

	void	**delete_layers**	(LayerPropertiesIterator iterators)	Deletes the layer properties nodes specified by the iterator
	void	**delete_layers**	(unsigned int index, LayerPropertiesIterator[] iterators)	Deletes the layer properties nodes specified by the iterator
	void	**descend**	(InstElement[] path, int index)	Descends further into the hierarchy.
[iter]	Annotation	**each_annotation**		Iterates over all annotations attached to this view
[const,iter]	Annotation	**each_annotation_selecte**		Iterate over each selected annotation objects, yielding a Annotation object for each of them
[iter]	Image	**each_image**		Iterate over all images attached to this view
[const,iter]	Image	**each_image_selected**		Iterate over each selected image object, yielding a Image object for each of them
[iter]	LayerPropertiesNodeRef	**each_layer**		Hierarchically iterates over the layers in the first layer list
[iter]	LayerPropertiesNode	**each_layer**	(unsigned int layer_list)	Hierarchically iterates over the layers in the given layer list
[const,iter]	ObjectInstPath	**each_object_selected**		Iterates over each selected geometrical object, yielding a ObjectInstPath object for each of them
[const,iter]	ObjectInstPath	**each_object_selected_tra**		Iterates over each geometrical objects in the transient selection, yielding a ObjectInstPath object for each of them
	void	**enable_edits**	(bool enable)	Enables or disables edits
[const]	LayerPropertiesIterat	**end_layers**		End iterator for the layers
[const]	LayerPropertiesIterator	**end_layers**	(unsigned int index)	End iterator for the layers
	void	**erase_annotation**	(int id)	Erases the annotation given by the id
	void	**erase_cellview**	(unsigned int index)	Erases the cellview with the given index
	void	**erase_image**	(unsigned long id)	Erase the given image
	void	**expand_layer_properties**		Expands the layer properties for all tabs
	void	**expand_layer_properties**	(unsigned int index)	Expands the layer properties for the given tab
[const]	string	**get_config**	(string name)	Gets the value of a local configuration parameter
	string[]	**get_config_names**		Gets the configuration parameter names
	QImage	**get_image**	(unsigned int width,	Gets the layout image as a QImage

			unsigned int height)	
	QImage	**get_image_with_options**	(unsigned int width, unsigned int height, int linewidth = 0, int oversampling = 0, double resolution = 0, const DBox target = current, bool monochrome = false)	Gets the layout image as a QImage (with options)
	string	**get_line_style**	(unsigned int index)	Gets the line style string for the style with the given index
	PixelBuffer	**get_pixels**	(unsigned int width, unsigned int height)	Gets the layout image as a PixelBuffer
	PixelBuffer	**get_pixels_with_options**	(unsigned int width, unsigned int height, int linewidth = 0, int oversampling = 0, double resolution = 0, const DBox target = current)	Gets the layout image as a PixelBuffer (with options)
	BitmapBuffer	**get_pixels_with_options_**	(unsigned int width, unsigned int height, int linewidth = 0, const DBox target = current)	Gets the layout image as a PixelBuffer (with options)
	QImage	**get_screenshot**		Gets a screenshot as a QImage
	PixelBuffer	**get_screenshot_pixels**		Gets a screenshot as a PixelBuffer
	string	**get_stipple**	(unsigned int index)	Gets the stipple pattern string for the pattern with the given index
[const]	bool	**has_annotation_selectio**		Returns true, if annotations (rulers) are selected in this view
[const]	bool	**has_image_selection?**		Returns true, if images are selected in this view
[const]	bool	**has_object_selection?**		Returns true, if geometrical objects (shapes or cell instances) are selected in this view
	bool	**has_selection?**		Indicates whether any objects are selected
[const]	bool	**has_transient_object_sel**		Returns true, if geometrical objects (shapes or cell instances) are selected in this view in the transient selection
	void	**hide_cell**	(unsigned int cell_index, int cv_index)	Hides the given cell for the given cellview

	PixelBuffer	**icon_for_layer**	(const LayerPropertiesIterator iter, unsigned int w, unsigned int h, double dpr, unsigned int di_off = 0, bool no_state = false)	Creates an icon pixmap for the given layer.
	Image	**image**	(unsigned long id)	Gets the image given by an ID
[const]	void	**init_layer_properties**	(LayerProperties props)	Fills the layer properties for a new layer
	void	**insert_annotation**	(Annotation ptr obj)	Inserts an annotation object into the given view
	void	**insert_image**	(Image obj)	Insert an image object into the given view
	LayerPropertiesNodeRef	**insert_layer**	(const LayerPropertiesIterator iter, const LayerProperties node = LayerProperties())	Inserts the given layer properties node into the list before the given position
	LayerPropertiesNode	**insert_layer**	(unsigned int index, const LayerPropertiesIterator iter, const LayerProperties node = LayerProperties())	Inserts the given layer properties node into the list before the given position
	void	**insert_layer_list**	(unsigned int index)	Inserts a new layer properties list at the given index
[const]	bool	**is_cell_hidden?**	(unsigned int cell_index, int cv_index)	Returns true, if the cell is hidden
	bool	**is_dirty?**		Gets a flag indicating whether one of the layouts displayed needs saving
[const]	bool	**is_editable?**		Returns true if the view is in editable mode
	bool	**is_transacting?**		Indicates if a transaction is ongoing
	LayoutToNetlist ptr	**l2ndb**	(int index)	Gets the netlist database with the given index
	void	**load_layer_props**	(string fn)	Loads the layer properties
	void	**load_layer_props**	(string fn, bool add_default)	Loads the layer properties with options
	void	**load_layer_props**	(string fn, int cv_index,	Loads the layer properties with options

			bool add_default)	
	unsigned int	**load_layout**	(string filename, const LoadLayoutOptions options, string technology, bool add_cellview = true)	Loads a (new) file into the layout view with the given technology
	unsigned int	**load_layout**	(string filename, const LoadLayoutOptions options, bool add_cellview = true)	Loads a (new) file into the layout view
	unsigned int	**load_layout**	(string filename, string technology, bool add_cellview = true)	Loads a (new) file into the layout view with the given technology
	unsigned int	**load_layout**	(string filename, bool add_cellview = true)	Loads a (new) file into the layout view
	LayoutVsSchematic ptr	**lvsdb**	(unsigned int index)	Gets the netlist database with the given index
	void	**max_hier**		Selects all hierarchy levels available
[const]	int	**max_hier_levels**		Returns the maximum hierarchy level up to which to display geometries
	void	**max_hier_levels=**	(int level)	Sets the maximum hierarchy level up to which to display geometries
	AbstractMenu ptr	**menu**		Gets the AbstractMenu associated with this view.
[const]	int	**min_hier_levels**		Returns the minimum hierarchy level at which to display geometries
	void	**min_hier_levels=**	(int level)	Sets the minimum hierarchy level at which to display geometries
[const]	string	**mode_name**		Gets the name of the current mode.
[const]	string[]	**mode_names**		Gets the names of the available modes.
	NetlistBrowserDialog ptr	**netlist_browser**		Gets the netlist browser object for the given layout view
[const]	unsigned int	**num_l2ndbs**		Gets the number of netlist databases loaded into this view
[const]	unsigned int	**num_layer_lists**		Gets the number of layer properties tabs present

[const]	unsigned int	**num_rdbs**		Gets the number of report databases loaded into this view
[const]	ObjectInstPath[]	**object_selection**		Returns a list of selected objects
[const]	void	**object_selection=**	(ObjectInstPath[] sel)	Sets the list of selected objects
[signal]	void	**on_active_cellview_changed**		An event indicating that the active cellview has changed
[signal]	void	**on_annotation_changed**	(int id)	A event indicating that an annotation has been modified
[signal]	void	**on_annotation_selection_changed**		A event indicating that the annotation selection has changed
[signal]	void	**on_annotations_changed**		A event indicating that annotations have been added or removed
[signal]	void	**on_apply_technology**	(int cellview_index)	An event indicating that a cellview has requested a new technology
[signal]	void	**on_cell_visibility_change**		An event indicating that the visibility of one or more cells has changed
[signal]	void	**on_cellview_changed**	(int cellview_index)	An event indicating that a cellview has changed
[signal]	void	**on_cellviews_changed**		An event indicating that the cellview collection has changed
[signal]	void	**on_close**		A event indicating that the view is about to close
[signal]	void	**on_current_layer_list_ch**	(int index)	An event indicating the current layer list (the selected tab) has changed
[signal]	void	**on_file_open**		An event indicating that a file was opened
[signal]	void	**on_hide**		A event indicating that the view is going to become invisible
[signal]	void	**on_image_changed**	(int id)	A event indicating that an image has been modified
[signal]	void	**on_image_selection_cha**		A event indicating that the image selection has changed
[signal]	void	**on_images_changed**		A event indicating that images have been added or removed
[signal]	void	**on_l2ndb_list_changed**		An event that is triggered the list of netlist databases is changed
[signal]	void	**on_layer_list_changed**	(int flags)	An event indicating that the layer list has changed
[signal]	void	**on_layer_list_deleted**	(int index)	An event indicating that a layer list (a tab) has been removed

[signal]	void	**on_layer_list_inserted**	(int index)	An event indicating that a layer list (a tab) has been inserted
[signal]	void	**on_rdb_list_changed**		An event that is triggered the list of report databases is changed
[signal]	void	**on_selection_changed**		An event that is triggered if the selection is changed
[signal]	void	**on_show**		A event indicating that the view is going to become visible
[signal]	void	**on_transient_selection_changed**		An event that is triggered if the transient selection is changed
[signal]	void	**on_viewport_changed**		An event indicating that the viewport (the visible rectangle) has changed
	D25View ptr	**open_d25_view**		Opens the 2.5d view window and returns a reference to the D25View object.
	void	**pan_center**	(const DPoint p)	Pans to the given point
	void	**pan_down**		Pans down
	void	**pan_left**		Pans to the left
	void	**pan_right**		Pans to the right
	void	**pan_up**		Pans upward
	ReportDatabase ptr	**rdb**	(int index)	Gets the report database with the given index
	void	**register_annotation_tem**	(const Annotation annotation, string title, int mode = RulerModeNormal)	Registers the given annotation as a template for this particular view
	void	**reload_layout**	(unsigned int cv)	Reloads the given cellview
	void	**remove_l2ndb**	(unsigned int index)	Removes a netlist database with the given index
	void	**remove_line_style**	(unsigned int index)	Removes the line style with the given index
	void	**remove_rdb**	(unsigned int index)	Removes a report database with the given index
	void	**remove_stipple**	(unsigned int index)	Removes the stipple pattern with the given index
	void	**remove_unused_layers**		Removes unused layers from layer list
	void	**rename_cellview**	(string name, int index)	Renames the cellview with the given index

void	rename_layer_list	(unsigned int index, string name)	Sets the title of the given layer properties tab
void	replace_annotation	(int id, const Annotation obj)	Replaces the annotation given by the id with the new one
void	replace_image	(unsigned long id, Image new_obj)	Replace an image object with the new image
unsigned int	replace_l2ndb	(unsigned int db_index, LayoutToNetlist ptr db)	Replaces the netlist database with the given index
void	replace_layer_node	(const LayerPropertiesIterator iter, const LayerProperties node)	Replaces the layer node at the position given by "iter" with a new one
void	replace_layer_node	(unsigned int index, const LayerPropertiesIterator iter, const LayerProperties node)	Replaces the layer node at the position given by "iter" with a new one
unsigned int	replace_lvsdb	(unsigned int db_index, LayoutVsSchematic ptr db)	Replaces the database with the given index
unsigned int	replace_rdb	(unsigned int db_index, ReportDatabase ptr db)	Replaces the report database with the given index
void	reset_title		Resets the title to the standard title
void	resize	(unsigned int w, unsigned int h)	Resizes the layout view to the given dimension
void	save_as	(unsigned int index, string filename, const SaveLayoutOptions options)	Saves a layout to the given stream file
void	save_image	(string filename, unsigned int width, unsigned int height)	Saves the layout as an image to the given file
void	save_image_with_option	(string filename, unsigned int width, unsigned int height, int linewidth = 0, int oversampling = 0, double resolution = 0,	Saves the layout as an image to the given file (with options)

			const DBox target = current, bool monochrome = false)	
	void	**save_layer_props**	(string fn)	Saves the layer properties
	void	**save_screenshot**	(string filename)	Saves a screenshot to the given file
	void	**select_all**		Selects all objects from the view
	void	**select_from**	(const DPoint point, LayoutView::SelectionM mode = Replace)	Selects the objects from a given point
	void	**select_from**	(const DBox box, LayoutView::SelectionMode mode = Replace)	Selects the objects from a given box
[const]	void	**select_object**	(const ObjectInstPath obj)	Adds the given selection to the list of selected objects
[const]	unsigned int[][]	**selected_cells_paths**	(int cv_index)	Gets the paths of the selected cells
[const]	LayerPropertiesIterat	**selected_layers**		Gets the selected layers
	DBox	**selection_bbox**		Returns the bounding box of the current selection
	unsigned long	**selection_size**		Returns the number of selected objects
	void	**send_enter_event**		Sends a mouse window leave event
	void	**send_key_press_event**	(unsigned int key, unsigned int buttons)	Sends a key press event
	void	**send_leave_event**		Sends a mouse window leave event
	void	**send_mouse_double_clic**	(const DPoint pt, unsigned int buttons)	Sends a mouse button double-click event
	void	**send_mouse_move_event**	(const DPoint pt, unsigned int buttons)	Sends a mouse move event
	void	**send_mouse_press_ever**	(const DPoint pt, unsigned int buttons)	Sends a mouse button press event
	void	**send_mouse_release_event**	(const DPoint pt, unsigned int buttons)	Sends a mouse button release event
	void	**send_wheel_event**	(int delta, bool horizontal, const DPoint pt, unsigned int buttons)	Sends a mouse wheel event
	void	**set_config**	(string name, string value)	Sets a local configuration parameter with the given name to the given value

	void	**set_layer_properties**	(const LayerPropertiesIterator iter, const LayerProperties props)	Sets the layer properties of the layer pointed to by the iterator
	void	**set_layer_properties**	(unsigned int index, const LayerPropertiesIterator iter, const LayerProperties props)	Sets the layer properties of the layer pointed to by the iterator
	void	**show_all_cells**		Makes all cells shown (cancel effects of hide_cell)
	void	**show_all_cells**	(int cv_index)	Makes all cells shown (cancel effects of hide_cell) for the specified cell view
	void	**show_cell**	(unsigned int cell_index, int cv_index)	Shows the given cell for the given cellview (cancel effect of hide_cell)
	void	**show_image**	(unsigned long id, bool visible)	Shows or hides the given image
	void	**show_l2ndb**	(int l2ndb_index, int cv_index)	Shows a netlist database in the marker browser on a certain layout
	unsigned int	**show_layout**	(Layout ptr layout, bool add_cellview)	Shows an existing layout in the view
	unsigned int	**show_layout**	(Layout ptr layout, string tech, bool add_cellview)	Shows an existing layout in the view
	unsigned int	**show_layout**	(Layout ptr layout, string tech, bool add_cellview, bool init_layers)	Shows an existing layout in the view
	void	**show_lvsdb**	(int lvsdb_index, int cv_index)	Shows a netlist database in the marker browser on a certain layout
	void	**show_rdb**	(int rdb_index, int cv_index)	Shows a report database in the marker browser on a certain layout
	void	**stop**		Stops redraw thread and close any browsers
	void	**stop_redraw**		Stops the redraw thread
	void	**switch_mode**	(string mode)	Switches the mode.
[const]	string	**title**		Returns the view's title string
	void	**title=**	(string title)	Sets the title of the view

	void	**transaction**	(string description)	Begins a transaction
	void	**transient_to_selection**		Turns the transient selection into the actual selection
	void	**unregister_annotation_templates**	(string category)	Unregisters the template or templates with the given category string on this particular view
[const]	void	**unselect_object**	(const ObjectInstPath obj)	Removes the given selection from the list of selected objects
	void	**update_content**		Updates the layout view to the current state
[const]	int	**viewport_height**		Return the viewport height in pixels
[const]	DCplxTrans	**viewport_trans**		Returns the transformation that converts micron coordinates to pixels
[const]	int	**viewport_width**		Returns the viewport width in pixels
	QWidget ptr	**widget**		Gets the QWidget object of the view
	void	**zoom_box**	(const DBox box)	Sets the viewport to the given box
	void	**zoom_fit**		Fits the contents of the current view into the window
	void	**zoom_fit_sel**		Fits the contents of the current selection into the window
	void	**zoom_in**		Zooms in somewhat
	void	**zoom_out**		Zooms out somewhat

Public static methods and constants

[static,const]	LayoutView::SelectionMod	**Add**	Adds to any existing selection
[static,const]	LayoutView::SelectionMode	**Invert**	Adds to any existing selection, if it's not there yet or removes it from the selection if it's already selected
[static,const]	unsigned int	**LV_Naked**	With this option, no separate views will be provided
[static,const]	unsigned int	**LV_NoBookmarksView**	With this option, no bookmarks view will be provided (see bookmarks_frame)
[static,const]	unsigned int	**LV_NoEditorOptionsP**	With this option, no editor options panel will be provided (see editor_options_frame)
[static,const]	unsigned int	**LV_NoGrid**	With this option, the grid background is not shown
[static,const]	unsigned int	**LV_NoHierarchyPanel**	With this option, no cell hierarchy view will be provided (see hierarchy_control_frame)

[static,const]	unsigned int	**LV_NoLayers**	With this option, no layers view will be provided (see layer_control_frame)
[static,const]	unsigned int	**LV_NoLibrariesView**	With this option, no library view will be provided (see libraries_frame)
[static,const]	unsigned int	**LV_NoMove**	With this option, move operations are not supported
[static,const]	unsigned int	**LV_NoPlugins**	With this option, all plugins are disabled
[static,const]	unsigned int	**LV_NoPropertiesPopup**	This option disables the properties popup on double click
[static,const]	unsigned int	**LV_NoSelection**	With this option, objects cannot be selected
[static,const]	unsigned int	**LV_NoServices**	This option disables all services except the ones for pure viewing
[static,const]	unsigned int	**LV_NoTracker**	With this option, mouse position tracking is not supported
[static,const]	unsigned int	**LV_NoZoom**	With this option, zooming is disabled
[static,const]	LayoutView::SelectionMod	**Replace**	Replaces the existing selection
[static,const]	LayoutView::SelectionMode	**Reset**	Removes from any existing selection
	LayoutView ptr	**current**	Returns the current view
	string[]	**menu_symbols**	Gets all available menu symbols (see call_menu).

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	unsigned int[]	**get_current_cell_path**	(int cv_index)	Use of this method is deprecated
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
	void	**save_as**	(unsigned int index, string filename, bool gzip, const SaveLayoutOptions options)	Use of this method is deprecated
	void	**select_cell**	(unsigned int cell_index, int cv_index)	Use of this method is deprecated

void	select_cell_path	(unsigned int[] cell_index, int cv_index)	Use of this method is deprecated
void	set_active_cellview_inde:	(int index)	Use of this method is deprecated. Use active_setview_index= instead
void	set_current_cell_path	(int cv_index, unsigned int[] cell_path)	Use of this method is deprecated
void	set_current_layer_list	(unsigned int index)	Use of this method is deprecated. Use current_layer_list= instead
void	set_title	(string title)	Use of this method is deprecated. Use title= instead

Detailed description

Add

Signature: *[static,const]* LayoutView::SelectionMode **Add**

Description: Adds to any existing selection

Python specific notes:
The object exposes a readable attribute 'Add'. This is the getter.

Invert

Signature: *[static,const]* LayoutView::SelectionMode **Invert**

Description: Adds to any existing selection, if it's not there yet or removes it from the selection if it's already selected

Python specific notes:
The object exposes a readable attribute 'Invert'. This is the getter.

LV_Naked

Signature: *[static,const]* unsigned int **LV_Naked**

Description: With this option, no separate views will be provided

Use this value with the constructor's 'options' argument. This option is basically equivalent to using LV_NoLayers+LV_NoHierarchyPanel+LV_NoLibrariesView+LV_NoBookmarksView

This constant has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'LV_Naked'. This is the getter.

LV_NoBookmarksView

Signature: *[static,const]* unsigned int **LV_NoBookmarksView**

Description: With this option, no bookmarks view will be provided (see bookmarks_frame)

Use this value with the constructor's 'options' argument.

This constant has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'LV_NoBookmarksView'. This is the getter.

LV_NoEditorOptionsPanel

Signature: *[static,const]* unsigned int **LV_NoEditorOptionsPanel**

Description: With this option, no editor options panel will be provided (see editor_options_frame)

Use this value with the constructor's 'options' argument.

This constant has been introduced in version 0.27.

Python specific notes:

	The object exposes a readable attribute 'LV_NoEditorOptionsPanel'. This is the getter.
LV_NoGrid	**Signature**: *[static,const]* unsigned int **LV_NoGrid** **Description**: With this option, the grid background is not shown Use this value with the constructor's 'options' argument. This constant has been introduced in version 0.27. **Python specific notes:** The object exposes a readable attribute 'LV_NoGrid'. This is the getter.
LV_NoHierarchyPanel	**Signature**: *[static,const]* unsigned int **LV_NoHierarchyPanel** **Description**: With this option, no cell hierarchy view will be provided (see hierarchy_control_frame) Use this value with the constructor's 'options' argument. This constant has been introduced in version 0.27. **Python specific notes:** The object exposes a readable attribute 'LV_NoHierarchyPanel'. This is the getter.
LV_NoLayers	**Signature**: *[static,const]* unsigned int **LV_NoLayers** **Description**: With this option, no layers view will be provided (see layer_control_frame) Use this value with the constructor's 'options' argument. This constant has been introduced in version 0.27. **Python specific notes:** The object exposes a readable attribute 'LV_NoLayers'. This is the getter.
LV_NoLibrariesView	**Signature**: *[static,const]* unsigned int **LV_NoLibrariesView** **Description**: With this option, no library view will be provided (see libraries_frame) Use this value with the constructor's 'options' argument. This constant has been introduced in version 0.27. **Python specific notes:** The object exposes a readable attribute 'LV_NoLibrariesView'. This is the getter.
LV_NoMove	**Signature**: *[static,const]* unsigned int **LV_NoMove** **Description**: With this option, move operations are not supported Use this value with the constructor's 'options' argument. This constant has been introduced in version 0.27. **Python specific notes:** The object exposes a readable attribute 'LV_NoMove'. This is the getter.
LV_NoPlugins	**Signature**: *[static,const]* unsigned int **LV_NoPlugins** **Description**: With this option, all plugins are disabled Use this value with the constructor's 'options' argument. This constant has been introduced in version 0.27. **Python specific notes:** The object exposes a readable attribute 'LV_NoPlugins'. This is the getter.

LV_NoPropertiesPopup

Signature: *[static,const]* unsigned int **LV_NoPropertiesPopup**

Description: This option disables the properties popup on double click

Use this value with the constructor's 'options' argument.

This constant has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'LV_NoPropertiesPopup'. This is the getter.

LV_NoSelection

Signature: *[static,const]* unsigned int **LV_NoSelection**

Description: With this option, objects cannot be selected

Use this value with the constructor's 'options' argument.

This constant has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'LV_NoSelection'. This is the getter.

LV_NoServices

Signature: *[static,const]* unsigned int **LV_NoServices**

Description: This option disables all services except the ones for pure viewing

Use this value with the constructor's 'options' argument. With this option, all manipulation features are disabled, except zooming. It is equivalent to LV_NoMove + LV_NoTracker + LV_NoSelection + LV_NoPlugins.

This constant has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'LV_NoServices'. This is the getter.

LV_NoTracker

Signature: *[static,const]* unsigned int **LV_NoTracker**

Description: With this option, mouse position tracking is not supported

Use this value with the constructor's 'options' argument. This option is not useful currently as no mouse tracking support is provided.

This constant has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'LV_NoTracker'. This is the getter.

LV_NoZoom

Signature: *[static,const]* unsigned int **LV_NoZoom**

Description: With this option, zooming is disabled

Use this value with the constructor's 'options' argument.

This constant has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'LV_NoZoom'. This is the getter.

Replace

Signature: *[static,const]* LayoutView::SelectionMode **Replace**

Description: Replaces the existing selection

Python specific notes:
The object exposes a readable attribute 'Replace'. This is the getter.

Reset

Signature: *[static,const]* LayoutView::SelectionMode **Reset**

Description: Removes from any existing selection

Python specific notes:

The object exposes a readable attribute 'Reset'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

active_cellview

Signature: CellView **active_cellview**

Description: Gets the active cellview (shown in hierarchy browser)

This is a convenience method which is equivalent to cellview(active_cellview_index()).

This method has been introduced in version 0.19. Starting from version 0.25, the returned object can be manipulated which will have an immediate effect on the display.

active_cellview_index

Signature: *[const]* int **active_cellview_index**

Description: Gets the index of the active cellview (shown in hierarchy browser)

active_setview_index=

Signature: void **active_setview_index=** (int index)

Description: Makes the cellview with the given index the active one (shown in hierarchy browser)

See active_cellview_index.

This method has been renamed from set_active_cellview_index to active_cellview_index= in version 0.25. The original name is still available, but is deprecated.

Python specific notes:
The object exposes a writable attribute 'active_setview_index'. This is the setter.

add_l2ndb

Signature: unsigned int **add_l2ndb** (LayoutToNetlist ptr db)

Description: Adds the given netlist database to the view

Returns: The index of the database within the view (see l2ndb)

This method will add an existing database to the view. It will then appear in the netlist database browser. A similar method is create_l2ndb which will create a new database within the view.

This method has been added in version 0.26.

add_line_style

(1) Signature: unsigned int **add_line_style** (string name, unsigned int data, unsigned int bits)

Description: Adds a custom line style

name: The name under which this pattern will appear in the style editor

data: A bit set with the new line style pattern (bit 0 is the leftmost pixel)

bits: The number of bits to be used

Returns: The index of the newly created style, which can be used as the line style index of LayerProperties.

This method has been introduced in version 0.25.

(2) Signature: unsigned int **add_line_style** (string name, string string)

Description: Adds a custom line style from a string

name: The name under which this pattern will appear in the style editor

string: A string describing the bits of the pattern ('.' for missing pixel, '*' for a set pixel)

Returns: The index of the newly created style, which can be used as the line style index of LayerProperties.

This method has been introduced in version 0.25.

add_lvsdb

Signature: unsigned int **add_lvsdb** (LayoutVsSchematic ptr db)

Description: Adds the given database to the view

Returns: The index of the database within the view (see lvsdb)

This method will add an existing database to the view. It will then appear in the netlist database browser. A similar method is create_lvsdb which will create a new database within the view.

This method has been added in version 0.26.

add_missing_layers

Signature: void **add_missing_layers**

Description: Adds new layers to layer list

This method was introduced in version 0.19.

add_rdb

Signature: unsigned int **add_rdb** (ReportDatabase ptr db)

Description: Adds the given report database to the view

Returns: The index of the database within the view (see rdb)

This method will add an existing database to the view. It will then appear in the marker database browser. A similar method is create_rdb which will create a new database within the view.

This method has been added in version 0.26.

add_stipple

(1) Signature: unsigned int **add_stipple** (string name, unsigned int[] data, unsigned int bits)

Description: Adds a stipple pattern

name: The name under which this pattern will appear in the stipple editor

data: See above

bits: See above

Returns: The index of the newly created stipple pattern, which can be used as the dither pattern index of LayerProperties.

'data' is an array of unsigned integers describing the bits that make up the stipple pattern. If the array has less than 32 entries, the pattern will be repeated vertically. The number of bits used can be less than 32 bit which can be specified by the 'bits' parameter. Logically, the pattern will be put at the end of the list.

(2) Signature: unsigned int **add_stipple** (string name, string string)

Description: Adds a stipple pattern given by a string

name: The name under which this pattern will appear in the stipple editor

string: See above

Returns: The index of the newly created stipple pattern, which can be used as the dither pattern index of LayerProperties.

'string' is a string describing the pattern. It consists of one or more lines composed of '.' or '*' characters and separated by newline characters. A '.' is for a missing pixel and '*' for a set pixel. The length of each line must be the same. Blanks before or after each line are ignored.

This method has been introduced in version 0.25.

annotation

Signature: Annotation **annotation** (int id)

Description: Gets the annotation given by an ID

Returns a reference to the annotation given by the respective ID or an invalid annotation if the ID is not valid. Use Annotation#is_valid? to determine whether the returned annotation is valid or not.

The returned annotation is a 'live' object and changing it will update the view.

This method has been introduced in version 0.25.

annotation_templates

Signature: variant[][] **annotation_templates**

Description: Gets a list of Annotation objects representing the annotation templates.

Annotation templates are the rulers available in the ruler drop-down (preset ruler types). This method will fetch the templates available. This method returns triplets '(annotation, title, mode)'. The first member of the triplet is the annotation object representing the template. The second member is the title string displayed in the menu for this templates. The third member is the mode value (one of the RulerMode... constants - e.g RulerModeNormal).

The positions of the returned annotation objects are undefined.

This method has been introduced in version 0.28.

ascend

Signature: InstElement **ascend** (int index)

Description: Ascends upwards in the hierarchy.

Removes one element from the specific path of the cellview with the given index. Returns the element removed.

begin_layers

(1) Signature: *[const]* LayerPropertiesIterator **begin_layers**

Description: Begin iterator for the layers

This iterator delivers the layers of this view, either in a recursive or non-recursive fashion, depending which iterator increment methods are used. The iterator delivered by end_layers is the past-the-end iterator. It can be compared against a current iterator to check, if there are no further elements.

Starting from version 0.25, an alternative solution is provided with 'each_layer' which is based on the LayerPropertiesNodeRef class.

(2) Signature: *[const]* LayerPropertiesIterator **begin_layers** (unsigned int index)

Description: Begin iterator for the layers

This iterator delivers the layers of this view, either in a recursive or non-recursive fashion, depending which iterator increment methods are used. The iterator delivered by end_layers is the past-the-end iterator. It can be compared against a current iterator to check, if there are no further elements. This version addresses a specific list in a multi-tab layer properties arrangement with the "index" parameter. This method has been introduced in version 0.21.

bookmark_view

Signature: void **bookmark_view** (string name)

Description: Bookmarks the current view under the given name

name: The name under which to bookmark the current state

box

Signature: *[const]* DBox **box**

Description: Returns the displayed box in micron space

call_menu

Signature: void **call_menu** (string symbol)

Description: Calls the menu item with the provided symbol.

To obtain all symbols, use menu_symbols.

This method has been introduced in version 0.27.

cancel

Signature: void **cancel**

Description: Cancels all edit operations

This method will stop all pending edit operations (i.e. drag and drop) and cancel the current selection. Calling this method is useful to ensure there are no potential interactions with the script's functionality.

cellview

Signature: CellView **cellview** (unsigned int cv_index)

Description: Gets the cellview object for a given index

cv_index: The cellview index for which to get the object for

Starting with version 0.25, this method returns a CellView object that can be manipulated to directly reflect any changes in the display.

cellviews

Signature: *[const]* unsigned int **cellviews**

Description: Gets the number of cellviews

clear_annotations

Signature: void **clear_annotations**

Description: Clears all annotations on this view

clear_config

Signature: void **clear_config**

Description: Clears the local configuration parameters

See set_config for a description of the local configuration parameters.

clear_images

Signature: void **clear_images**

Description: Clear all images on this view

clear_layers

(1) Signature: void **clear_layers**

Description: Clears all layers

(2) Signature: void **clear_layers** (unsigned int index)

Description: Clears all layers for the given layer properties list

This version addresses a specific list in a multi-tab layer properties arrangement with the "index" parameter. This method has been introduced in version 0.21.

clear_line_styles

Signature: void **clear_line_styles**

Description: Removes all custom line styles

All line styles except the fixed ones are removed. If any of the custom styles is still used by the layers displayed, the results will be undefined. This method has been introduced in version 0.25.

clear_object_selection

Signature: *[const]* void **clear_object_selection**

Description: Clears the selection of geometrical objects (shapes or cell instances)

The selection of other objects (such as annotations and images) will not be affected.

This method has been introduced in version 0.24

clear_selection

Signature: void **clear_selection**

Description: Clears the selection of all objects (shapes, annotations, images ...)

This method has been introduced in version 0.26.2

clear_stipples

Signature: void **clear_stipples**

Description: Removes all custom line styles

All stipple pattern except the fixed ones are removed. If any of the custom stipple pattern is still used by the layers displayed, the results will be undefined.

clear_transactions

Signature: void **clear_transactions**

Description: Clears all transactions

Discard all actions in the undo buffer. After clearing that buffer, no undo is available. It is important to clear the buffer when making database modifications outside transactions, i.e after that modifications have been done. If failing to do so, 'undo' operations are likely to produce invalid results. This method was introduced in version 0.16.

clear_transient_selection

Signature: void **clear_transient_selection**

Description: Clears the transient selection (mouse-over hightlights) of all objects (shapes, annotations, images ...)

This method has been introduced in version 0.26.2

close

Signature: void **close**

Description: Closes the view

This method has been added in version 0.27.

commit

Signature: void **commit**

Description: Ends a transaction

See transaction for a detailed description of transactions. This method was introduced in version 0.16.

commit_config

Signature: void **commit_config**

Description: Commits the configuration settings

Some configuration options are queued for performance reasons and become active only after 'commit_config' has been called. After a sequence of set_config calls, this method should be called to activate the settings made by these calls.

This method has been introduced in version 0.25.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

create_l2ndb

Signature: unsigned int **create_l2ndb** (string name)

Description: Creates a new netlist database and returns the index of the new database

name:	The name of the new netlist database
Returns:	The index of the new database

This method returns an index of the new netlist database. Use l2ndb to get the actual object. If a netlist database with the given name already exists, a unique name will be created. The name will be replaced by the file name when a file is loaded into the netlist database.

This method has been added in version 0.26.

create_layout

(1) Signature: unsigned int **create_layout** (bool add_cellview)

Description: Creates a new, empty layout

Returns: The index of the cellview created.

The add_cellview parameter controls whether to create a new cellview (true) or clear all cellviews before (false).

This version will associate the new layout with the default technology.

(2) Signature: unsigned int **create_layout** (string tech, bool add_cellview)

Description: Create a new, empty layout and associate it with the given technology

Returns: The index of the cellview created.

The add_cellview parameter controls whether to create a new cellview (true) or clear all cellviews before (false).

This variant has been introduced in version 0.22.

(3) Signature: unsigned int **create_layout** (string tech, bool add_cellview, bool init_layers)

Description: Create a new, empty layout and associate it with the given technology

Returns: The index of the cellview created.

The add_cellview parameter controls whether to create a new cellview (true) or clear all cellviews before (false). This variant also allows one to control whether the layer properties are initialized (init_layers = true) or not (init_layers = false).

This variant has been introduced in version 0.22.

create_lvsdb

Signature: unsigned int **create_lvsdb** (string name)

Description: Creates a new netlist database and returns the index of the new database

name: The name of the new netlist database

Returns: The index of the new database

This method returns an index of the new netlist database. Use lvsdb to get the actual object. If a netlist database with the given name already exists, a unique name will be created. The name will be replaced by the file name when a file is loaded into the netlist database.

This method has been added in version 0.26.

create_measure_ruler

Signature: Annotation **create_measure_ruler** (const DPoint point, int ac = Annotation#AngleAny)

Description: Createas an auto-measure ruler at the given point.

point: The seed point where to create the auto-measure ruler

ac: The orientation constraints (determines the search direction too)

Returns: The new ruler object

The ac parameters takes one of the Angle... constants from Annotation.

This method will create a ruler with a measurement, looking to the sides of the seed point for visible layout in the vicinity. The angle constraint determines the main directions where to look.

If suitable edges are found, the method will pull a line between the closest edges. The ruler's endpoints will sit on these lines and the ruler's length will be the distance. Only visible layers will participate in the measurement.

The new ruler is inserted into the view already. It is created with the default style of rulers. If the measurement fails because there is no layout in the vicinity, a ruler with identical start and end points will be created.

This method was introduced in version 0.26.

create_rdb

Signature: unsigned int **create_rdb** (string name)

Description: Creates a new report database and returns the index of the new database

name:	The name of the new report database
Returns:	The index of the new database

This method returns an index of the new report database. Use rdb to get the actual object. If a report database with the given name already exists, a unique name will be created. The name will be replaced by the file name when a file is loaded into the report database.

current

Signature: *[static]* LayoutView ptr **current**

Description: Returns the current view

The current view is the one that is shown in the current tab. Returns nil if no layout is loaded.

This method has been introduced in version 0.23.

current_layer

Signature: *[const]* LayerPropertiesIterator **current_layer**

Description: Gets the current layer view

Returns the LayerPropertiesIterator pointing to the current layer view (the one that has the focus). If no layer view is active currently, a null iterator is returned.

Python specific notes:
The object exposes a readable attribute 'current_layer'. This is the getter.

current_layer=

Signature: void **current_layer=** (const LayerPropertiesIterator iter)

Description: Sets the current layer view

Specifies an LayerPropertiesIterator pointing to the new current layer view.

This method has been introduced in version 0.23.

Python specific notes:
The object exposes a writable attribute 'current_layer'. This is the setter.

current_layer_list

Signature: *[const]* unsigned int **current_layer_list**

Description: Gets the index of the currently selected layer properties tab

This method has been introduced in version 0.21.

Python specific notes:
The object exposes a readable attribute 'current_layer_list'. This is the getter.

current_layer_list=

Signature: void **current_layer_list=** (unsigned int index)

Description: Sets the index of the currently selected layer properties tab

This method has been introduced in version 0.21.

Python specific notes:

	The object exposes a writable attribute 'current_layer_list'. This is the setter.
delete_layer	**(1) Signature**: void **delete_layer** (LayerPropertiesIterator iter) **Description**: Deletes the layer properties node specified by the iterator This method deletes the object that the iterator points to and invalidates the iterator since the object that the iterator points to is no longer valid. **(2) Signature**: void **delete_layer** (unsigned int index, LayerPropertiesIterator iter) **Description**: Deletes the layer properties node specified by the iterator This method deletes the object that the iterator points to and invalidates the iterator since the object that the iterator points to is no longer valid. This version addresses a specific list in a multi-tab layer properties arrangement with the "index" parameter. This method has been introduced in version 0.21.
delete_layer_list	**Signature**: void **delete_layer_list** (unsigned int index) **Description**: Deletes the given properties list At least one layer properties list must remain. This method may change the current properties list. This method has been introduced in version 0.21.
delete_layers	**(1) Signature**: void **delete_layers** (LayerPropertiesIterator[] iterators) **Description**: Deletes the layer properties nodes specified by the iterator This method deletes the nodes specifies by the iterators. This method is the most convenient way to delete multiple entries. This method has been added in version 0.22. **(2) Signature**: void **delete_layers** (unsigned int index, LayerPropertiesIterator[] iterators) **Description**: Deletes the layer properties nodes specified by the iterator This method deletes the nodes specifies by the iterators. This method is the most convenient way to delete multiple entries. This version addresses a specific list in a multi-tab layer properties arrangement with the "index" parameter. This method has been introduced in version 0.22.
descend	**Signature**: void **descend** (InstElement[] path, int index) **Description**: Descends further into the hierarchy. Adds the given path (given as an array of InstElement objects) to the specific path of the cellview with the given index. In effect, the cell addressed by the terminal of the new path components can be shown in the context of the upper cells, if the minimum hierarchy level is set to a negative value. The path is assumed to originate from the current cell and contain specific instances sorted from top to bottom.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

each_annotation

Signature: *[iter]* Annotation **each_annotation**

Description: Iterates over all annotations attached to this view

each_annotation_selected

Signature: *[const,iter]* Annotation **each_annotation_selected**

Description: Iterate over each selected annotation objects, yielding a Annotation object for each of them

This method was introduced in version 0.19.

each_image

Signature: *[iter]* Image **each_image**

Description: Iterate over all images attached to this view

With version 0.25, the objects returned by the iterator are references and can be manipulated to change their appearance.

each_image_selected

Signature: *[const,iter]* Image **each_image_selected**

Description: Iterate over each selected image object, yielding a Image object for each of them

This method was introduced in version 0.19.

each_layer

(1) Signature: *[iter]* LayerPropertiesNodeRef **each_layer**

Description: Hierarchically iterates over the layers in the first layer list

This iterator will recursively deliver the layers in the first layer list of the view. The objects presented by the iterator are LayerPropertiesNodeRef objects. They can be manipulated to apply changes to the layer settings or even the hierarchy of layers:

```
RBA::LayoutViewBase::current.each_layer do |lref|
  # lref is a RBA::LayerPropertiesNodeRef object
  lref.visible = false
end
```

This method was introduced in version 0.25.

(2) Signature: *[iter]* LayerPropertiesNodeRef **each_layer** (unsigned int layer_list)

Description: Hierarchically iterates over the layers in the given layer list

This version of this method allows specification of the layer list to be iterated over. The layer list is specified by its index which is a value between 0 and num_layer_lists-1.For details see the parameter-less version of this method.

This method was introduced in version 0.25.

each_object_selected

Signature: *[const,iter]* ObjectInstPath **each_object_selected**

Description: Iterates over each selected geometrical object, yielding a ObjectInstPath object for each of them

This iterator will deliver const objects - they cannot be modified. In order to modify the selection, create a copy of the ObjectInstPath objects, modify them and install the new selection using select_object or object_selection=.

Another way of obtaining the selection is object_selection, which returns an array of ObjectInstPath objects.

each_object_selected_transient

Signature: *[const,iter]* ObjectInstPath **each_object_selected_transient**

Description: Iterates over each geometrical objects in the transient selection, yielding a ObjectInstPath object for each of them

This method was introduced in version 0.18.

enable_edits

Signature: void **enable_edits** (bool enable)

Description: Enables or disables edits

enable: Enable edits if set to true

This method allows putting the view into read-only mode by disabling all edit functions. For doing so, this method has to be called with a 'false' argument. Calling it with a 'true' parameter enables all edits again. This method must not be confused with the edit/viewer mode. The LayoutView's enable_edits method is intended to temporarily disable all menu entries and functions which could allow the user to alter the database. In 0.25, this method has been moved from MainWindow to LayoutView.

end_layers

(1) Signature: *[const]* LayerPropertiesIterator **end_layers**

Description: End iterator for the layers

See begin_layers for a description about this iterator

(2) Signature: *[const]* LayerPropertiesIterator **end_layers** (unsigned int index)

Description: End iterator for the layers

See begin_layers for a description about this iterator This version addresses a specific list in a multi-tab layer properties arrangement with the "index" parameter. This method has been introduced in version 0.21.

erase_annotation

Signature: void **erase_annotation** (int id)

Description: Erases the annotation given by the id

Deletes an existing annotation given by the id parameter. The id of an annotation can be obtained through Annotation#id.

This method has been introduced in version 0.24. Starting with version 0.25, the annotation's Annotation#delete method can also be used to delete an annotation.

erase_cellview

Signature: void **erase_cellview** (unsigned int index)

Description: Erases the cellview with the given index

This closes the given cellview and unloads the layout associated with it, unless referred to by another cellview.

erase_image

Signature: void **erase_image** (unsigned long id)

Description: Erase the given image

id: The id of the object to erase

Erases the image with the given Id. The Id can be obtained with if "id" method of the image object.

This method has been introduced in version 0.20.

With version 0.25, Image#delete can be used to achieve the same results.

expand_layer_properties

(1) Signature: void **expand_layer_properties**

Description: Expands the layer properties for all tabs

This method will expand all wildcard specifications in the layer properties by iterating over the specified objects (i.e. layers, cellviews) and by replacing default colors and stipples by the ones specified with the palettes.

This method was introduced in version 0.21.

(2) Signature: void **expand_layer_properties** (unsigned int index)

Description: Expands the layer properties for the given tab

This method will expand all wildcard specifications in the layer properties by iterating over the specified objects (i.e. layers, cellviews) and by replacing default colors and stipples by the ones specified with the palettes.

This method was introduced in version 0.21.

get_config

Signature: *[const]* string **get_config** (string name)

Description: Gets the value of a local configuration parameter

name: The name of the configuration parameter whose value shall be obtained (a string)

Returns: The value of the parameter

See set_config for a description of the local configuration parameters.

get_config_names

Signature: string[] **get_config_names**

Description: Gets the configuration parameter names

Returns: A list of configuration parameter names

This method returns the names of all known configuration parameters. These names can be used to get and set configuration parameter values.

This method was introduced in version 0.25.

get_current_cell_path

Signature: *[const]* unsigned int[] **get_current_cell_path** (int cv_index)

Description: Gets the cell path of the current cell

cv_index: The cellview index for which to get the current path from (usually this will be the active cellview index)

Use of this method is deprecated

The current cell is the one highlighted in the browser with the focus rectangle. The current path is returned for the cellview given by cv_index. The cell path is a list of cell indices from the top cell to the current cell.

This method is was deprecated in version 0.25 since from then, the CellView object can be used to obtain an manipulate the selected cell.

get_image

Signature: QImage **get_image** (unsigned int width, unsigned int height)

Description: Gets the layout image as a QImage

width: The width of the image to render in pixel.

height: The height of the image to render in pixel.

The image contains the current scene (layout, annotations etc.). The image is drawn synchronously with the given width and height. Drawing may take some time.

get_image_with_options

Signature: QImage **get_image_with_options** (unsigned int width, unsigned int height, int linewidth = 0, int oversampling = 0, double resolution = 0, const DBox target = current, bool monochrome = false)

Description: Gets the layout image as a QImage (with options)

width: The width of the image to render in pixel.

height: The height of the image to render in pixel.

linewidth: The width of a line in pixels (usually 1) or 0 for default.

oversampling: The oversampling factor (1..3) or 0 for default.

resolution: The resolution (pixel size compared to a screen pixel size, i.e 1/oversampling) or 0 for default.

target_box: The box to draw or an empty box for default.

monochrome: If true, monochrome images will be produced.

The image contains the current scene (layout, annotations etc.). The image is drawn synchronously with the given width and height. Drawing may take some time. Monochrome images don't have background or annotation objects currently.

This method has been introduced in version 0.23.10.

get_line_style

Signature: string **get_line_style** (unsigned int index)

Description: Gets the line style string for the style with the given index

This method will return the line style string for the style with the given index. The format of the string is the same than the string accepted by add_line_style. An empty string corresponds to 'solid line'.

This method has been introduced in version 0.25.

get_pixels

Signature: PixelBuffer **get_pixels** (unsigned int width, unsigned int height)

Description: Gets the layout image as a PixelBuffer

width: The width of the image to render in pixel.

height: The height of the image to render in pixel.

The image contains the current scene (layout, annotations etc.). The image is drawn synchronously with the given width and height. Drawing may take some time. This method has been introduced in 0.28.

get_pixels_with_options

Signature: PixelBuffer **get_pixels_with_options** (unsigned int width, unsigned int height, int linewidth = 0, int oversampling = 0, double resolution = 0, const DBox target = current)

Description: Gets the layout image as a PixelBuffer (with options)

width: The width of the image to render in pixel.

height: The height of the image to render in pixel.

linewidth: The width of a line in pixels (usually 1) or 0 for default.

oversampling:	The oversampling factor (1..3) or 0 for default.
resolution:	The resolution (pixel size compared to a screen pixel size, i.e 1/oversampling) or 0 for default.
target_box:	The box to draw or an empty box for default.

The image contains the current scene (layout, annotations etc.). The image is drawn synchronously with the given width and height. Drawing may take some time. This method has been introduced in 0.28.

get_pixels_with_options_mon

Signature: BitmapBuffer **get_pixels_with_options_mono** (unsigned int width, unsigned int height, int linewidth = 0, const DBox target = current)

Description: Gets the layout image as a PixelBuffer (with options)

width:	The width of the image to render in pixel.
height:	The height of the image to render in pixel.
linewidth:	The width of a line in pixels (usually 1) or 0 for default.
target_box:	The box to draw or an empty box for default.

The image contains the current scene (layout, annotations etc.). The image is drawn synchronously with the given width and height. Drawing may take some time. Monochrome images don't have background or annotation objects currently.

This method has been introduced in 0.28.

get_screenshot

Signature: QImage **get_screenshot**

Description: Gets a screenshot as a QImage

Getting the image requires the drawing to be complete. Ideally, synchronous mode is switched on for the application to guarantee this condition. The image will have the size of the viewport showing the current layout.

get_screenshot_pixels

Signature: PixelBuffer **get_screenshot_pixels**

Description: Gets a screenshot as a PixelBuffer

Getting the image requires the drawing to be complete. Ideally, synchronous mode is switched on for the application to guarantee this condition. The image will have the size of the viewport showing the current layout. This method has been introduced in 0.28.

get_stipple

Signature: string **get_stipple** (unsigned int index)

Description: Gets the stipple pattern string for the pattern with the given index

This method will return the stipple pattern string for the pattern with the given index. The format of the string is the same than the string accepted by add_stipple.

This method has been introduced in version 0.25.

has_annotation_selection?

Signature: *[const]* bool **has_annotation_selection?**

Description: Returns true, if annotations (rulers) are selected in this view

This method was introduced in version 0.19.

has_image_selection?

Signature: *[const]* bool **has_image_selection?**

Description: Returns true, if images are selected in this view

This method was introduced in version 0.19.

has_object_selection?

Signature: *[const]* bool **has_object_selection?**

Description: Returns true, if geometrical objects (shapes or cell instances) are selected in this view

has_selection?

Signature: bool **has_selection?**

Description: Indicates whether any objects are selected

This method has been introduced in version 0.27

has_transient_object_selectio

Signature: *[const]* bool **has_transient_object_selection?**

Description: Returns true, if geometrical objects (shapes or cell instances) are selected in this view in the transient selection

The transient selection represents the objects selected when the mouse hovers over the layout windows. This selection is not used for operations but rather to indicate which object would be selected if the mouse is clicked.

This method was introduced in version 0.18.

hide_cell

Signature: void **hide_cell** (unsigned int cell_index, int cv_index)

Description: Hides the given cell for the given cellview

icon_for_layer

Signature: PixelBuffer **icon_for_layer** (const LayerPropertiesIterator iter, unsigned int w, unsigned int h, double dpr, unsigned int di_off = 0, bool no_state = false)

Description: Creates an icon pixmap for the given layer.

The icon will have size w times h pixels multiplied by the device pixel ratio (dpr). The dpr is The number of physical pixels per logical pixels on high-DPI displays.

'di_off' will shift the dither pattern by the given number of (physical) pixels. If 'no_state' is true, the icon will not reflect visibility or validity states but rather the display style.

This method has been introduced in version 0.28.

image

Signature: Image **image** (unsigned long id)

Description: Gets the image given by an ID

Returns a reference to the image given by the respective ID or an invalid image if the ID is not valid. Use Image#is_valid? to determine whether the returned image is valid or not.

The returned image is a 'live' object and changing it will update the view.

This method has been introduced in version 0.25.

init_layer_properties

Signature: *[const]* void **init_layer_properties** (LayerProperties props)

Description: Fills the layer properties for a new layer

props: The layer properties object to initialize.

This method initializes a layer properties object's color and stipples according to the defaults for the given layer source specification. The layer's source must be set already on the layer properties object.

This method was introduced in version 0.19.

insert_annotation

Signature: void **insert_annotation** (Annotation ptr obj)

Description: Inserts an annotation object into the given view

Inserts a new annotation into the view. Existing annotation will remain. Use clear_annotations to delete them before inserting new ones. Use replace_annotation to replace an existing one with a new one. Starting with version 0.25 this method modifies self's ID to reflect the ID of the ruler created. After an annotation is inserted into the view, it can be modified and the changes of properties will become reflected immediately in the view.

insert_image

Signature: void **insert_image** (Image obj)

Description: Insert an image object into the given view

Insert the image object given by obj into the view.

With version 0.25, this method will attach the image object to the view and the image object will become a 'live' object - i.e. changes to the object will change the appearance of the image on the screen.

insert_layer

(1) Signature: LayerPropertiesNodeRef **insert_layer** (const LayerPropertiesIterator iter, const LayerProperties node = LayerProperties())

Description: Inserts the given layer properties node into the list before the given position

This method inserts the new properties node before the position given by "iter" and returns a const reference to the element created. The iterator that specified the position will remain valid after the node was inserted and will point to the newly created node. It can be used to add further nodes. To add children to the node inserted, use iter.last_child as insertion point for the next insert operations.

Since version 0.22, this method accepts LayerProperties and LayerPropertiesNode objects. A LayerPropertiesNode object can contain a hierarchy of further nodes. Since version 0.26 the node parameter is optional and the reference returned by this method can be used to set the properties of the new node.

(2) Signature: LayerPropertiesNodeRef **insert_layer** (unsigned int index, const LayerPropertiesIterator iter, const LayerProperties node = LayerProperties())

Description: Inserts the given layer properties node into the list before the given position

This version addresses a specific list in a multi-tab layer properties arrangement with the "index" parameter. This method inserts the new properties node before the position given by "iter" and returns a const reference to the element created. The iterator that specified the position will remain valid after the node was inserted and will point to the newly created node. It can be used to add further nodes. This method has been introduced in version 0.21. Since version 0.22, this method accepts LayerProperties and LayerPropertiesNode objects. A LayerPropertiesNode object can contain a hierarchy of further nodes. Since version 0.26 the node parameter is optional and the reference returned by this method can be used to set the properties of the new node.

insert_layer_list

Signature: void **insert_layer_list** (unsigned int index)

Description: Inserts a new layer properties list at the given index

This method inserts a new tab at the given position. The current layer properties list will be changed to the new list. This method has been introduced in version 0.21.

is_cell_hidden?

Signature: *[const]* bool **is_cell_hidden?** (unsigned int cell_index, int cv_index)

Description: Returns true, if the cell is hidden

Returns: True, if the cell with "cell_index" is hidden for the cellview "cv_index"

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_dirty?

Signature: bool **is_dirty?**

Description: Gets a flag indicating whether one of the layouts displayed needs saving

A layout is 'dirty' if it is modified and needs saving. This method returns true if this is the case for at least one of the layouts shown in the view.

This method has been introduced in version 0.29.

is_editable?

Signature: *[const]* bool **is_editable?**

Description: Returns true if the view is in editable mode

This read-only attribute has been added in version 0.27.5.

is_transacting?

Signature: bool **is_transacting?**

Description: Indicates if a transaction is ongoing

See transaction for a detailed description of transactions. This method was introduced in version 0.16.

l2ndb

Signature: LayoutToNetlist ptr **l2ndb** (int index)

Description: Gets the netlist database with the given index

Returns: The LayoutToNetlist object or nil if the index is not valid

This method has been added in version 0.26.

load_layer_props

(1) Signature: void **load_layer_props** (string fn)

Description: Loads the layer properties

fn: The file name of the .lyp file to load

Load the layer properties from the file given in "fn"

(2) Signature: void **load_layer_props** (string fn, bool add_default)

Description: Loads the layer properties with options

fn: The file name of the .lyp file to load

add_default: If true, default layers will be added for each other layer in the layout

Load the layer properties from the file given in "fn". This version allows one to specify whether defaults should be used for all other layers by setting "add_default" to true.

This variant has been added on version 0.21.

(3) Signature: void **load_layer_props** (string fn, int cv_index, bool add_default)

Description: Loads the layer properties with options

fn: The file name of the .lyp file to load

cv_index: See description text

add_default: If true, default layers will be added for each other layer in the layout

Load the layer properties from the file given in "fn". This version allows one to specify whether defaults should be used for all other layers by setting "add_default" to true. It can be used to load the layer properties for a specific cellview by setting "cv_index" to the index for which the layer properties file should be applied. All present definitions for this layout will be removed before the properties file is loaded. "cv_index" can be set to -1. In that case, the layer properties file is applied to each of the layouts individually.

Note that this version will override all cellview index definitions in the layer properties file.

This variant has been added on version 0.21.

load_layout

(1) Signature: unsigned int **load_layout** (string filename, const LoadLayoutOptions options, string technology, bool add_cellview = true)

Description: Loads a (new) file into the layout view with the given technology

Returns: The index of the cellview loaded.

Loads the file given by the "filename" parameter and associates it with the given technology. The options specify various options for reading the file. The add_cellview param controls whether to create a new cellview (true) or clear all cellviews before (false).

This version has been introduced in version 0.22. The 'add_cellview' argument has been made optional in version 0.28.

(2) Signature: unsigned int **load_layout** (string filename, const LoadLayoutOptions options, bool add_cellview = true)

Description: Loads a (new) file into the layout view

Returns: The index of the cellview loaded.

Loads the file given by the "filename" parameter. The options specify various options for reading the file. The add_cellview param controls whether to create a new cellview (true) or clear all cellviews before (false).

This method has been introduced in version 0.18. The 'add_cellview' argument has been made optional in version 0.28.

(3) Signature: unsigned int **load_layout** (string filename, string technology, bool add_cellview = true)

Description: Loads a (new) file into the layout view with the given technology

Returns: The index of the cellview loaded.

Loads the file given by the "filename" parameter and associates it with the given technology. The add_cellview param controls whether to create a new cellview (true) or clear all cellviews before (false).

This version has been introduced in version 0.22. The 'add_cellview' argument has been made optional in version 0.28.

(4) Signature: unsigned int **load_layout** (string filename, bool add_cellview = true)

Description: Loads a (new) file into the layout view

Returns: The index of the cellview loaded. The 'add_cellview' argument has been made optional in version 0.28.

Loads the file given by the "filename" parameter. The add_cellview param controls whether to create a new cellview (true) or clear all cellviews before (false).

lvsdb

Signature: LayoutVsSchematic ptr **lvsdb** (unsigned int index)

Description: Gets the netlist database with the given index

Returns: The LayoutVsSchematic object or nil if the index is not valid

This method has been added in version 0.26.

max_hier

Signature: void **max_hier**

Description: Selects all hierarchy levels available

Show the layout in full depth down to the deepest level of hierarchy. This method may cause a redraw.

max_hier_levels

Signature: *[const]* int **max_hier_levels**

Description: Returns the maximum hierarchy level up to which to display geometries

Returns: The maximum level up to which to display geometries

Python specific notes:
The object exposes a readable attribute 'max_hier_levels'. This is the getter.

max_hier_levels=

Signature: void **max_hier_levels=** (int level)

Description: Sets the maximum hierarchy level up to which to display geometries

level: The maximum level below which to display something

This methods allows setting the maximum hierarchy below which to display geometries.This method may cause a redraw if required.

Python specific notes:
The object exposes a writable attribute 'max_hier_levels'. This is the setter.

menu

Signature: AbstractMenu ptr **menu**

Description: Gets the AbstractMenu associated with this view.

In normal UI application mode this is the main window's view. For a detached view or in non-UI applications this is the view's private menu.

This method has been introduced in version 0.28.

menu_symbols

Signature: *[static]* string[] **menu_symbols**

Description: Gets all available menu symbols (see call_menu).

NOTE: currently this method delivers a superset of all available symbols. Depending on the context, no all symbols may trigger actual functionality.

This method has been introduced in version 0.27.

min_hier_levels

Signature: *[const]* int **min_hier_levels**

Description: Returns the minimum hierarchy level at which to display geometries

Returns: The minimum level at which to display geometries

Python specific notes:
The object exposes a readable attribute 'min_hier_levels'. This is the getter.

min_hier_levels=

Signature: void **min_hier_levels=** (int level)

Description: Sets the minimum hierarchy level at which to display geometries

level: The minimum level above which to display something

This methods allows setting the minimum hierarchy level above which to display geometries.This method may cause a redraw if required.

Python specific notes:
The object exposes a writable attribute 'min_hier_levels'. This is the setter.

mode_name

Signature: *[const]* string **mode_name**

Description: Gets the name of the current mode.

See switch_mode about a method to change the mode and mode_names for a method to retrieve all available mode names.

This method has been introduced in version 0.28.

mode_names

Signature: *[const]* string[] **mode_names**

Description: Gets the names of the available modes.

This method allows asking the view for the available mode names for switch_mode and for the value returned by mode.

This method has been introduced in version 0.28.

netlist_browser

Signature: NetlistBrowserDialog ptr **netlist_browser**

Description: Gets the netlist browser object for the given layout view

This method has been added in version 0.27.

new

Signature: *[static]* new LayoutView ptr **new** (bool editable = false, Manager ptr manager = nil, unsigned int options = 0)

Description: Creates a standalone view

editable:	True to make the view editable
manager:	The Manager object to enable undo/redo
options:	A combination of the values in the LV_... constants from LayoutViewBase

This constructor is for special purposes only. To create a view in the context of a main window, use MainWindow#create_view and related methods.

This constructor has been introduced in version 0.25. It has been enhanced with the arguments in version 0.27.

Python specific notes:
This method is the default initializer of the object.

num_l2ndbs

Signature: *[const]* unsigned int **num_l2ndbs**

Description: Gets the number of netlist databases loaded into this view

Returns: The number of LayoutToNetlist objects present in this view

This method has been added in version 0.26.

num_layer_lists

Signature: *[const]* unsigned int **num_layer_lists**

Description: Gets the number of layer properties tabs present

This method has been introduced in version 0.23.

num_rdbs

Signature: *[const]* unsigned int **num_rdbs**

Description: Gets the number of report databases loaded into this view

Returns: The number of ReportDatabase objects present in this view

object_selection

Signature: *[const]* ObjectInstPath[] **object_selection**

Description: Returns a list of selected objects

This method will deliver an array of ObjectInstPath objects listing the selected geometrical objects. Other selected objects such as annotations and images will not be contained in that list.

The list returned is an array of copies of ObjectInstPath objects. They can be modified, but they will become a new selection only after re-introducing them into the view through object_selection= or select_object.

Another way of obtaining the selected objects is each_object_selected.

This method has been introduced in version 0.24.

Python specific notes:
The object exposes a readable attribute 'object_selection'. This is the getter.

object_selection=

Signature: *[const]* void **object_selection=** (ObjectInstPath[] sel)

Description: Sets the list of selected objects

This method will set the selection of geometrical objects such as shapes and instances. It is the setter which complements the object_selection method.

Another way of setting the selection is through clear_object_selection and select_object.

This method has been introduced in version 0.24.

Python specific notes:
The object exposes a writable attribute 'object_selection'. This is the setter.

on_active_cellview_changed

Signature: *[signal]* void **on_active_cellview_changed**

Description: An event indicating that the active cellview has changed

If the active cellview is changed by selecting a new one from the drop-down list, this event is triggered. When this event is triggered, the cellview has already been changed. Before version 0.25 this event was based on the observer pattern obsolete now. The corresponding methods (add_active_cellview_changed/remove_active_cellview_changed) have been removed in 0.25.

Python specific notes:
The object exposes a readable attribute 'on_active_cellview_changed'. This is the getter.
The object exposes a writable attribute 'on_active_cellview_changed'. This is the setter.

on_annotation_changed

Signature: *[signal]* void **on_annotation_changed** (int id)

Description: A event indicating that an annotation has been modified

The argument of the event is the ID of the annotation that was changed. This event has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'on_annotation_changed'. This is the getter.
The object exposes a writable attribute 'on_annotation_changed'. This is the setter.

on_annotation_selection_cha

Signature: *[signal]* void **on_annotation_selection_changed**

Description: A event indicating that the annotation selection has changed

This event has been added in version 0.25.

	Python specific notes: The object exposes a readable attribute 'on_annotation_selection_changed'. This is the getter. The object exposes a writable attribute 'on_annotation_selection_changed'. This is the setter.
on_annotations_changed	**Signature**: *[signal]* void **on_annotations_changed** **Description**: A event indicating that annotations have been added or removed This event has been added in version 0.25. **Python specific notes:** The object exposes a readable attribute 'on_annotations_changed'. This is the getter. The object exposes a writable attribute 'on_annotations_changed'. This is the setter.
on_apply_technology	**Signature**: *[signal]* void **on_apply_technology** (int cellview_index) **Description**: An event indicating that a cellview has requested a new technology If the technology of a cellview is changed, this event is triggered. The integer parameter of this event will indicate the cellview that has changed. This event has been introduced in version 0.28. **Python specific notes:** The object exposes a readable attribute 'on_apply_technology'. This is the getter. The object exposes a writable attribute 'on_apply_technology'. This is the setter.
on_cell_visibility_changed	**Signature**: *[signal]* void **on_cell_visibility_changed** **Description**: An event indicating that the visibility of one or more cells has changed This event is triggered after the visibility of one or more cells has changed. Before version 0.25 this event was based on the observer pattern obsolete now. The corresponding methods (add_cell_visibility_observer/remove_cell_visibility_observer) have been removed in 0.25. **Python specific notes:** The object exposes a readable attribute 'on_cell_visibility_changed'. This is the getter. The object exposes a writable attribute 'on_cell_visibility_changed'. This is the setter.
on_cellview_changed	**Signature**: *[signal]* void **on_cellview_changed** (int cellview_index) **Description**: An event indicating that a cellview has changed If a cellview is modified, this event is triggered. When this event is triggered, the cellview have already been changed. The integer parameter of this event will indicate the cellview that has changed. Before version 0.25 this event was based on the observer pattern obsolete now. The corresponding methods (add_cellview_observer/remove_cellview_observer) have been removed in 0.25. **Python specific notes:** The object exposes a readable attribute 'on_cellview_changed'. This is the getter. The object exposes a writable attribute 'on_cellview_changed'. This is the setter.
on_cellviews_changed	**Signature**: *[signal]* void **on_cellviews_changed** **Description**: An event indicating that the cellview collection has changed If new cellviews are added or cellviews are removed, this event is triggered. When this event is triggered, the cellviews have already been changed. Before version 0.25 this event was based on the observer pattern obsolete now. The corresponding methods (add_cellview_list_observer/ remove_cellview_list_observer) have been removed in 0.25.

Python specific notes:
The object exposes a readable attribute 'on_cellviews_changed'. This is the getter.
The object exposes a writable attribute 'on_cellviews_changed'. This is the setter.

on_close

Signature: *[signal]* void **on_close**

Description: A event indicating that the view is about to close

This event is triggered when the view is going to be closed entirely.

It has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'on_close'. This is the getter.
The object exposes a writable attribute 'on_close'. This is the setter.

on_current_layer_list_changed

Signature: *[signal]* void **on_current_layer_list_changed** (int index)

Description: An event indicating the current layer list (the selected tab) has changed

index: The index of the new current layer list

This event is triggered after the current layer list was changed - i.e. a new tab was selected.

This event was introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'on_current_layer_list_changed'. This is the getter.
The object exposes a writable attribute 'on_current_layer_list_changed'. This is the setter.

on_file_open

Signature: *[signal]* void **on_file_open**

Description: An event indicating that a file was opened

If a file is loaded, this event is triggered. When this event is triggered, the file was already loaded and the new file is the new active cellview. Despite its name, this event is also triggered if a layout object is loaded into the view.

Before version 0.25 this event was based on the observer pattern obsolete now. The corresponding methods (add_file_open_observer/remove_file_open_observer) have been removed in 0.25.

Python specific notes:
The object exposes a readable attribute 'on_file_open'. This is the getter.
The object exposes a writable attribute 'on_file_open'. This is the setter.

on_hide

Signature: *[signal]* void **on_hide**

Description: A event indicating that the view is going to become invisible

It has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'on_hide'. This is the getter.
The object exposes a writable attribute 'on_hide'. This is the setter.

on_image_changed

Signature: *[signal]* void **on_image_changed** (int id)

Description: A event indicating that an image has been modified

The argument of the event is the ID of the image that was changed. This event has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'on_image_changed'. This is the getter.
The object exposes a writable attribute 'on_image_changed'. This is the setter.

on_image_selection_changed

Signature: *[signal]* void **on_image_selection_changed**

Description: A event indicating that the image selection has changed

This event has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'on_image_selection_changed'. This is the getter.
The object exposes a writable attribute 'on_image_selection_changed'. This is the setter.

on_images_changed

Signature: *[signal]* void **on_images_changed**

Description: A event indicating that images have been added or removed

This event has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'on_images_changed'. This is the getter.
The object exposes a writable attribute 'on_images_changed'. This is the setter.

on_l2ndb_list_changed

Signature: *[signal]* void **on_l2ndb_list_changed**

Description: An event that is triggered the list of netlist databases is changed

If a netlist database is added or removed, this event is triggered.

This method has been added in version 0.26.

Python specific notes:
The object exposes a readable attribute 'on_l2ndb_list_changed'. This is the getter.
The object exposes a writable attribute 'on_l2ndb_list_changed'. This is the setter.

on_layer_list_changed

Signature: *[signal]* void **on_layer_list_changed** (int flags)

Description: An event indicating that the layer list has changed

This event is triggered after the layer list has changed its configuration. The integer argument gives a hint about the nature of the changed: Bit 0 is set, if the properties (visibility, color etc.) of one or more layers have changed. Bit 1 is set if the hierarchy has changed. Bit 2 is set, if layer names have changed. Before version 0.25 this event was based on the observer pattern obsolete now. The corresponding methods (add_layer_list_observer/remove_layer_list_observer) have been removed in 0.25.

Python specific notes:
The object exposes a readable attribute 'on_layer_list_changed'. This is the getter.
The object exposes a writable attribute 'on_layer_list_changed'. This is the setter.

on_layer_list_deleted

Signature: *[signal]* void **on_layer_list_deleted** (int index)

Description: An event indicating that a layer list (a tab) has been removed

index: The index of the layer list that was removed

This event is triggered after the layer list has been removed - i.e. a tab was deleted.

This event was introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'on_layer_list_deleted'. This is the getter.
The object exposes a writable attribute 'on_layer_list_deleted'. This is the setter.

on_layer_list_inserted

Signature: *[signal]* void **on_layer_list_inserted** (int index)

Description: An event indicating that a layer list (a tab) has been inserted

index: The index of the layer list that was inserted

	This event is triggered after the layer list has been inserted - i.e. a new tab was created. This event was introduced in version 0.25. **Python specific notes:** The object exposes a readable attribute 'on_layer_list_inserted'. This is the getter. The object exposes a writable attribute 'on_layer_list_inserted'. This is the setter.
on_rdb_list_changed	**Signature**: *[signal]* void **on_rdb_list_changed** **Description**: An event that is triggered the list of report databases is changed If a report database is added or removed, this event is triggered. This event was translated from the Observer pattern to an event in version 0.25. **Python specific notes:** The object exposes a readable attribute 'on_rdb_list_changed'. This is the getter. The object exposes a writable attribute 'on_rdb_list_changed'. This is the setter.
on_selection_changed	**Signature**: *[signal]* void **on_selection_changed** **Description**: An event that is triggered if the selection is changed If the selection changed, this event is triggered. This event was translated from the Observer pattern to an event in version 0.25. **Python specific notes:** The object exposes a readable attribute 'on_selection_changed'. This is the getter. The object exposes a writable attribute 'on_selection_changed'. This is the setter.
on_show	**Signature**: *[signal]* void **on_show** **Description**: A event indicating that the view is going to become visible It has been added in version 0.25. **Python specific notes:** The object exposes a readable attribute 'on_show'. This is the getter. The object exposes a writable attribute 'on_show'. This is the setter.
on_transient_selection_chang	**Signature**: *[signal]* void **on_transient_selection_changed** **Description**: An event that is triggered if the transient selection is changed If the transient selection is changed, this event is triggered. The transient selection is the highlighted selection when the mouse hovers over some object(s). This event was translated from the Observer pattern to an event in version 0.25. **Python specific notes:** The object exposes a readable attribute 'on_transient_selection_changed'. This is the getter. The object exposes a writable attribute 'on_transient_selection_changed'. This is the setter.
on_viewport_changed	**Signature**: *[signal]* void **on_viewport_changed** **Description**: An event indicating that the viewport (the visible rectangle) has changed This event is triggered after a new display rectangle was chosen - for example, because the user zoomed into the layout. Before version 0.25 this event was based on the observer pattern obsolete now. The corresponding methods (add_viewport_changed_observer/remove_viewport_changed_observer) have been removed in 0.25. **Python specific notes:** The object exposes a readable attribute 'on_viewport_changed'. This is the getter.

The object exposes a writable attribute 'on_viewport_changed'. This is the setter.

open_d25_view

Signature: D25View ptr **open_d25_view**
Description: Opens the 2.5d view window and returns a reference to the D25View object.
This method has been introduced in version 0.28.

pan_center

Signature: void **pan_center** (const DPoint p)
Description: Pans to the given point
The window is positioned such that "p" becomes the new center

pan_down

Signature: void **pan_down**
Description: Pans down

pan_left

Signature: void **pan_left**
Description: Pans to the left

pan_right

Signature: void **pan_right**
Description: Pans to the right

pan_up

Signature: void **pan_up**
Description: Pans upward

rdb

Signature: ReportDatabase ptr **rdb** (int index)
Description: Gets the report database with the given index
Returns: The ReportDatabase object or nil if the index is not valid

register_annotation_template

Signature: void **register_annotation_template** (const Annotation annotation, string title, int mode = RulerModeNormal)
Description: Registers the given annotation as a template for this particular view
title: The title to use for the ruler template
mode: The mode the ruler will be created in (see Ruler... constants)
@annotation The annotation to use for the template (positions are ignored)
See Annotation#register_template for a method doing the same on application level. This method is hardly useful normally, but can be used when customizing layout views as individual widgets.
This method has been added in version 0.28.

reload_layout

Signature: void **reload_layout** (unsigned int cv)
Description: Reloads the given cellview
cv: The index of the cellview to reload

remove_l2ndb

Signature: void **remove_l2ndb** (unsigned int index)
Description: Removes a netlist database with the given index
The: index of the netlist database to remove from this view

This method has been added in version 0.26.

remove_line_style

Signature: void **remove_line_style** (unsigned int index)

Description: Removes the line style with the given index

The line styles with an index less than the first custom style. If a style is removed that is still used, the results are undefined.

This method has been introduced in version 0.25.

remove_rdb

Signature: void **remove_rdb** (unsigned int index)

Description: Removes a report database with the given index

The: index of the report database to remove from this view

remove_stipple

Signature: void **remove_stipple** (unsigned int index)

Description: Removes the stipple pattern with the given index

The pattern with an index less than the first custom pattern cannot be removed. If a stipple pattern is removed that is still used, the results are undefined.

remove_unused_layers

Signature: void **remove_unused_layers**

Description: Removes unused layers from layer list

This method was introduced in version 0.19.

rename_cellview

Signature: void **rename_cellview** (string name, int index)

Description: Renames the cellview with the given index

If the name is not unique, a unique name will be constructed from the name given. The name may be different from the filename but is associated with the layout object. If a layout is shared between multiple cellviews (which may happen due to a clone of the layout view for example), all cellviews are renamed.

rename_layer_list

Signature: void **rename_layer_list** (unsigned int index, string name)

Description: Sets the title of the given layer properties tab

This method has been introduced in version 0.21.

replace_annotation

Signature: void **replace_annotation** (int id, const Annotation obj)

Description: Replaces the annotation given by the id with the new one

Replaces an existing annotation given by the id parameter with the new one. The id of an annotation can be obtained through Annotation#id.

This method has been introduced in version 0.24.

replace_image

Signature: void **replace_image** (unsigned long id, Image new_obj)

Description: Replace an image object with the new image

id: The id of the object to replace

new_obj: The new object to replace the old one

Replaces the image with the given Id with the new object. The Id can be obtained with if "id" method of the image object.

This method has been introduced in version 0.20.

replace_l2ndb

Signature: unsigned int **replace_l2ndb** (unsigned int db_index, LayoutToNetlist ptr db)

Description: Replaces the netlist database with the given index

Returns: The index of the database within the view (see lvsdb)

If the index is not valid, the database will be added to the view (see add_lvsdb).

This method has been added in version 0.26.

replace_layer_node

(1) Signature: void **replace_layer_node** (const LayerPropertiesIterator iter, const LayerProperties node)

Description: Replaces the layer node at the position given by "iter" with a new one

Since version 0.22, this method accepts LayerProperties and LayerPropertiesNode objects. A LayerPropertiesNode object can contain a hierarchy of further nodes.

(2) Signature: void **replace_layer_node** (unsigned int index, const LayerPropertiesIterator iter, const LayerProperties node)

Description: Replaces the layer node at the position given by "iter" with a new one

This version addresses a specific list in a multi-tab layer properties arrangement with the "index" parameter. This method has been introduced in version 0.21. Since version 0.22, this method accepts LayerProperties and LayerPropertiesNode objects. A LayerPropertiesNode object can contain a hierarchy of further nodes.

replace_lvsdb

Signature: unsigned int **replace_lvsdb** (unsigned int db_index, LayoutVsSchematic ptr db)

Description: Replaces the database with the given index

Returns: The index of the database within the view (see lvsdb)

If the index is not valid, the database will be added to the view (see add_lvsdb).

This method has been added in version 0.26.

replace_rdb

Signature: unsigned int **replace_rdb** (unsigned int db_index, ReportDatabase ptr db)

Description: Replaces the report database with the given index

Returns: The index of the database within the view (see rdb)

If the index is not valid, the database will be added to the view (see add_rdb).

This method has been added in version 0.26.

reset_title

Signature: void **reset_title**

Description: Resets the title to the standard title

See set_title and title for a description about how titles are handled.

resize

Signature: void **resize** (unsigned int w, unsigned int h)

Description: Resizes the layout view to the given dimension

This method has been made available in all builds in 0.28.

save_as

(1) Signature: void **save_as** (unsigned int index, string filename, bool gzip, const SaveLayoutOptions options)

Description: Saves a layout to the given stream file

index:	The cellview index of the layout to save.
filename:	The file to write.
gzip:	Ignored.
options:	Writer options.

Use of this method is deprecated

The layout with the given index is written to the stream file with the given options. 'options' is a SaveLayoutOptions object that specifies which format to write and further options such as scaling factor etc. Calling this method is equivalent to calling 'write' on the respective layout object.

This method is deprecated starting from version 0.23. The compression mode is determined from the file name automatically and the gzip parameter is ignored.

(2) Signature: void **save_as** (unsigned int index, string filename, const SaveLayoutOptions options)

Description: Saves a layout to the given stream file

index:	The cellview index of the layout to save.
filename:	The file to write.
options:	Writer options.

The layout with the given index is written to the stream file with the given options. 'options' is a SaveLayoutOptions object that specifies which format to write and further options such as scaling factor etc. Calling this method is equivalent to calling 'write' on the respective layout object.

If the file name ends with a suffix ".gz" or ".gzip", the file is compressed with the zlib algorithm.

save_image

Signature: void **save_image** (string filename, unsigned int width, unsigned int height)

Description: Saves the layout as an image to the given file

filename:	The file to which to write the screenshot to.
width:	The width of the image to render in pixel.
height:	The height of the image to render in pixel.

The image contains the current scene (layout, annotations etc.). The image is written as a PNG file to the given file. The image is drawn synchronously with the given width and height. Drawing may take some time.

save_image_with_options

Signature: void **save_image_with_options** (string filename, unsigned int width, unsigned int height, int linewidth = 0, int oversampling = 0, double resolution = 0, const DBox target = current, bool monochrome = false)

Description: Saves the layout as an image to the given file (with options)

filename:	The file to which to write the screenshot to.
width:	The width of the image to render in pixel.
height:	The height of the image to render in pixel.
linewidth:	The line width scale factor (usually 1) or 0 for 1/resolution.
oversampling:	The oversampling factor (1..3) or 0 for the oversampling the view was configured with.
resolution:	The resolution (pixel size compared to a screen pixel) or 0 for 1/oversampling.
target_box:	The box to draw or an empty box for default.
monochrome:	If true, monochrome images will be produced.

The image contains the current scene (layout, annotations etc.). The image is written as a PNG file to the given file. The image is drawn synchronously with the given width and height. Drawing may take some time. Monochrome images don't have background or annotation objects currently.

The 'linewidth' factor scales the layout style line widths.

The 'oversampling' factor will use multiple passes passes to create a single image pixels. An oversampling factor of 2 uses 2x2 virtual pixels to generate an output pixel. This results in a smoother image. This however comes with a corresponding memory and run time penalty. When using oversampling, you can set linewidth and resolution to 0. This way, line widths and stipple pattern are scaled such that the resulting image is equivalent to the standard image.

The 'resolution' is the pixel size used to translate font sizes and stipple pattern. A resolution of 0.5 renders twice as large fonts and stipple pattern. When combining this value with an oversampling factor of 2 and a line width factor of 2, the resulting image is an oversampled version of the standard image.

Examples:

```
# standard image 500x500 pixels (oversampling as configured in the view)
layout_view.save_image_with_options("image.png", 500, 500)

# 2x oversampled image with 500x500 pixels
layout_view.save_image_with_options("image.png", 500, 500, 0, 2, 0)

# 2x scaled image with 1000x1000 pixels
layout_view.save_image_with_options("image.png", 1000, 1000, 2, 1, 0.5)
```

This method has been introduced in 0.23.10.

save_layer_props

Signature: void **save_layer_props** (string fn)

Description: Saves the layer properties

Save the layer properties to the file given in "fn"

save_screenshot

Signature: void **save_screenshot** (string filename)

Description: Saves a screenshot to the given file

filename: The file to which to write the screenshot to.

The screenshot is written as a PNG file to the given file. This requires the drawing to be complete. Ideally, synchronous mode is switched on for the application to guarantee this condition. The image will have the size of the viewport showing the current layout.

select_all

Signature: void **select_all**

Description: Selects all objects from the view

This method has been introduced in version 0.27

select_cell

Signature: void **select_cell** (unsigned int cell_index, int cv_index)

Description: Selects a cell by index for a certain cell view

Use of this method is deprecated

Select the current (top) cell by specifying a path (a list of cell indices from top to the actual cell) and the cellview index for which this cell should become the currently shown one. This method selects the cell to be drawn. In constrast, the set_current_cell_path method selects the cell that is

	highlighted in the cell tree (but not necessarily drawn). This method is was deprecated in version 0.25 since from then, the CellView object can be used to obtain an manipulate the selected cell.
select_cell_path	**Signature**: void **select_cell_path** (unsigned int[] cell_index, int cv_index) **Description**: Selects a cell by cell index for a certain cell view Use of this method is deprecated Select the current (top) cell by specifying a cell indexand the cellview index for which this cell should become the currently shown one. The path to the cell is constructed by selecting one that leads to a top cell. This method selects the cell to be drawn. In constrast, the set_current_cell_path method selects the cell that is highlighted in the cell tree (but not necessarily drawn). This method is was deprecated in version 0.25 since from then, the CellView object can be used to obtain an manipulate the selected cell.
select_from	**(1) Signature**: void **select_from** (const DPoint point, LayoutView::SelectionMode mode = Replace) **Description**: Selects the objects from a given point The mode indicates whether to add to the selection, replace the selection, remove from selection or invert the selected status of the objects found around the given point. This method has been introduced in version 0.27 **(2) Signature**: void **select_from** (const DBox box, LayoutView::SelectionMode mode = Replace) **Description**: Selects the objects from a given box The mode indicates whether to add to the selection, replace the selection, remove from selection or invert the selected status of the objects found inside the given box. This method has been introduced in version 0.27
select_object	**Signature**: *[const]* void **select_object** (const ObjectInstPath obj) **Description**: Adds the given selection to the list of selected objects The selection provided by the ObjectInstPath descriptor is added to the list of selected objects. To clear the previous selection, use clear_object_selection. The selection of other objects (such as annotations and images) will not be affected. Another way of selecting objects is object_selection=. This method has been introduced in version 0.24
selected_cells_paths	**Signature**: *[const]* unsigned int[][] **selected_cells_paths** (int cv_index) **Description**: Gets the paths of the selected cells Gets a list of cell paths to the cells selected in the cellview given by cv_index. The "selected cells" are the ones selected in the cell list or cell tree. This is not the "current cell" which is the one that is shown in the layout window. The cell paths are arrays of cell indexes where the last element is the actual cell selected. This method has be introduced in version 0.25.
selected_layers	**Signature**: *[const]* LayerPropertiesIterator[] **selected_layers** **Description**: Gets the selected layers

	Returns an array of LayerPropertiesIterator objects pointing to the currently selected layers. If no layer view is selected currently, an empty array is returned.
selection_bbox	**Signature**: DBox **selection_bbox** **Description**: Returns the bounding box of the current selection This method has been introduced in version 0.26.2
selection_size	**Signature**: unsigned long **selection_size** **Description**: Returns the number of selected objects This method has been introduced in version 0.27
send_enter_event	**Signature**: void **send_enter_event** **Description**: Sends a mouse window leave event This method is intended to emulate the mouse mouse window leave events sent by Qt normally in environments where Qt is not present. This method was introduced in version 0.28.
send_key_press_event	**Signature**: void **send_key_press_event** (unsigned int key, unsigned int buttons) **Description**: Sends a key press event This method is intended to emulate the key press events sent by Qt normally in environments where Qt is not present. The arguments follow the conventions used within Plugin#key_event for example. This method was introduced in version 0.28.
send_leave_event	**Signature**: void **send_leave_event** **Description**: Sends a mouse window leave event This method is intended to emulate the mouse mouse window leave events sent by Qt normally in environments where Qt is not present. This method was introduced in version 0.28.
send_mouse_double_clicked_event	**Signature**: void **send_mouse_double_clicked_event** (const DPoint pt, unsigned int buttons) **Description**: Sends a mouse button double-click event This method is intended to emulate the mouse button double-click events sent by Qt normally in environments where Qt is not present. The arguments follow the conventions used within Plugin#mouse_moved_event for example. This method was introduced in version 0.28.
send_mouse_move_event	**Signature**: void **send_mouse_move_event** (const DPoint pt, unsigned int buttons) **Description**: Sends a mouse move event This method is intended to emulate the mouse move events sent by Qt normally in environments where Qt is not present. The arguments follow the conventions used within Plugin#mouse_moved_event for example. This method was introduced in version 0.28.
send_mouse_press_event	**Signature**: void **send_mouse_press_event** (const DPoint pt, unsigned int buttons) **Description**: Sends a mouse button press event

This method is intended to emulate the mouse button press events sent by Qt normally in environments where Qt is not present. The arguments follow the conventions used within Plugin#mouse_moved_event for example.

This method was introduced in version 0.28.

send_mouse_release_event

Signature: void **send_mouse_release_event** (const DPoint pt, unsigned int buttons)

Description: Sends a mouse button release event

This method is intended to emulate the mouse button release events sent by Qt normally in environments where Qt is not present. The arguments follow the conventions used within Plugin#mouse_moved_event for example.

This method was introduced in version 0.28.

send_wheel_event

Signature: void **send_wheel_event** (int delta, bool horizontal, const DPoint pt, unsigned int buttons)

Description: Sends a mouse wheel event

This method is intended to emulate the mouse wheel events sent by Qt normally in environments where Qt is not present. The arguments follow the conventions used within Plugin#wheel_event for example.

This method was introduced in version 0.28.

set_active_cellview_index

Signature: void **set_active_cellview_index** (int index)

Description: Makes the cellview with the given index the active one (shown in hierarchy browser)

Use of this method is deprecated. Use active_setview_index= instead

See active_cellview_index.

This method has been renamed from set_active_cellview_index to active_cellview_index= in version 0.25. The original name is still available, but is deprecated.

Python specific notes:
The object exposes a writable attribute 'active_setview_index'. This is the setter.

set_config

Signature: void **set_config** (string name, string value)

Description: Sets a local configuration parameter with the given name to the given value

name:	The name of the configuration parameter to set
value:	The value to which to set the configuration parameter

This method sets a local configuration parameter with the given name to the given value. Values can only be strings. Numerical values have to be converted into strings first. Local configuration parameters override global configurations for this specific view. This allows for example to override global settings of background colors. Any local settings are not written to the configuration file.

set_current_cell_path

Signature: void **set_current_cell_path** (int cv_index, unsigned int[] cell_path)

Description: Sets the path to the current cell

cv_index:	The cellview index for which to set the current path for (usually this will be the active cellview index)
path:	The path to the current cell

Use of this method is deprecated

The current cell is the one highlighted in the browser with the focus rectangle. The cell given by the path is highlighted and scrolled into view. To select the cell to be drawn, use the select_cell or select_cell_path method.

This method is was deprecated in version 0.25 since from then, the CellView object can be used to obtain an manipulate the selected cell.

set_current_layer_list

Signature: void **set_current_layer_list** (unsigned int index)

Description: Sets the index of the currently selected layer properties tab

Use of this method is deprecated. Use current_layer_list= instead

This method has been introduced in version 0.21.

Python specific notes:
The object exposes a writable attribute 'current_layer_list'. This is the setter.

set_layer_properties

(1) Signature: void **set_layer_properties** (const LayerPropertiesIterator iter, const LayerProperties props)

Description: Sets the layer properties of the layer pointed to by the iterator

This method replaces the layer properties of the element pointed to by "iter" by the properties given by "props". It will not change the hierarchy but just the properties of the given node.

(2) Signature: void **set_layer_properties** (unsigned int index, const LayerPropertiesIterator iter, const LayerProperties props)

Description: Sets the layer properties of the layer pointed to by the iterator

This method replaces the layer properties of the element pointed to by "iter" by the properties given by "props" in the tab given by "index". It will not change the hierarchy but just the properties of the given node.This version addresses a specific list in a multi-tab layer properties arrangement with the "index" parameter. This method has been introduced in version 0.21.

set_title

Signature: void **set_title** (string title)

Description: Sets the title of the view

title: The title string to use

Use of this method is deprecated. Use title= instead

Override the standard title of the view indicating the file names loaded by the specified title string. The title string can be reset with reset_title to the standard title again.

Python specific notes:
The object exposes a writable attribute 'title'. This is the setter.

show_all_cells

(1) Signature: void **show_all_cells**

Description: Makes all cells shown (cancel effects of hide_cell)

(2) Signature: void **show_all_cells** (int cv_index)

Description: Makes all cells shown (cancel effects of hide_cell) for the specified cell view

Unlike show_all_cells, this method will only clear the hidden flag on the cell view selected by cv_index.

This variant has been added in version 0.25.

show_cell

Signature: void **show_cell** (unsigned int cell_index, int cv_index)

Description: Shows the given cell for the given cellview (cancel effect of hide_cell)

show_image

Signature: void **show_image** (unsigned long id, bool visible)

Description: Shows or hides the given image

id: The id of the object to show or hide

visible: True, if the image should be shown

Sets the visibility of the image with the given Id. The Id can be obtained with if "id" method of the image object.

This method has been introduced in version 0.20.

With version 0.25, Image#visible= can be used to achieve the same results.

show_l2ndb

Signature: void **show_l2ndb** (int l2ndb_index, int cv_index)

Description: Shows a netlist database in the marker browser on a certain layout

The netlist browser is opened showing the netlist database with the index given by "l2ndb_index". It will be attached (i.e. navigate to) the layout with the given cellview index in "cv_index".

This method has been added in version 0.26.

show_layout

(1) Signature: unsigned int **show_layout** (Layout ptr layout, bool add_cellview)

Description: Shows an existing layout in the view

Returns: The index of the cellview created.

Shows the given layout in the view. If add_cellview is true, the new layout is added to the list of cellviews in the view.

Note: once a layout is passed to the view with show_layout, it is owned by the view and must not be destroyed with the 'destroy' method.

This method has been introduced in version 0.22.

(2) Signature: unsigned int **show_layout** (Layout ptr layout, string tech, bool add_cellview)

Description: Shows an existing layout in the view

Returns: The index of the cellview created.

Shows the given layout in the view. If add_cellview is true, the new layout is added to the list of cellviews in the view. The technology to use for that layout can be specified as well with the 'tech' parameter. Depending on the definition of the technology, layer properties may be loaded for example. The technology string can be empty for the default technology.

Note: once a layout is passed to the view with show_layout, it is owned by the view and must not be destroyed with the 'destroy' method.

This method has been introduced in version 0.22.

(3) Signature: unsigned int **show_layout** (Layout ptr layout, string tech, bool add_cellview, bool init_layers)

Description: Shows an existing layout in the view

Returns: The index of the cellview created.

Shows the given layout in the view. If add_cellview is true, the new layout is added to the list of cellviews in the view. The technology to use for that layout can be specified as well with the 'tech' parameter. Depending on the definition of the technology, layer properties may be loaded for example. The technology string can be empty for the default technology. This variant also allows one to control whether the layer properties are initialized (init_layers = true) or not (init_layers = false).

Note: once a layout is passed to the view with show_layout, it is owned by the view and must not be destroyed with the 'destroy' method.

This method has been introduced in version 0.22.

show_lvsdb

Signature: void **show_lvsdb** (int lvsdb_index, int cv_index)

Description: Shows a netlist database in the marker browser on a certain layout

The netlist browser is opened showing the netlist database with the index given by "lvsdb_index". It will be attached (i.e. navigate to) the layout with the given cellview index in "cv_index".

This method has been added in version 0.26.

show_rdb

Signature: void **show_rdb** (int rdb_index, int cv_index)

Description: Shows a report database in the marker browser on a certain layout

The marker browser is opened showing the report database with the index given by "rdb_index". It will be attached (i.e. navigate to) the layout with the given cellview index in "cv_index".

stop

Signature: void **stop**

Description: Stops redraw thread and close any browsers

This method usually does not need to be called explicitly. The redraw thread is stopped automatically.

stop_redraw

Signature: void **stop_redraw**

Description: Stops the redraw thread

It is very important to stop the redraw thread before applying changes to the layout or the cell views and the LayoutView configuration. This is usually done automatically. For rare cases, where this is not the case, this method is provided.

switch_mode

Signature: void **switch_mode** (string mode)

Description: Switches the mode.

See mode_name about a method to get the name of the current mode and mode_names for a method to retrieve all available mode names.

This method has been introduced in version 0.28.

title

Signature: *[const]* string **title**

Description: Returns the view's title string

Returns: The title string

The title string is either a string composed of the file names loaded (in some "readable" manner) or a customized title string set by set_title.

Python specific notes:
The object exposes a readable attribute 'title'. This is the getter.

title=

Signature: void **title=** (string title)

Description: Sets the title of the view

title: The title string to use

Override the standard title of the view indicating the file names loaded by the specified title string. The title string can be reset with reset_title to the standard title again.

Python specific notes:

The object exposes a writable attribute 'title'. This is the setter.

transaction

Signature: void **transaction** (string description)

Description: Begins a transaction

description: A text that appears in the 'undo' description

A transaction brackets a sequence of database modifications that appear as a single undo action. Only modifications that are wrapped inside a transaction..commit call pair can be undone. Each transaction must be terminated with a commit method call, even if some error occurred. It is advisable therefore to catch errors and issue a commit call in this case.

This method was introduced in version 0.16.

transient_to_selection

Signature: void **transient_to_selection**

Description: Turns the transient selection into the actual selection

The current selection is cleared before. All highlighted objects under the mouse will become selected. This applies to all types of objects (rulers, shapes, images ...).

This method has been introduced in version 0.26.2

unregister_annotation_templ

Signature: void **unregister_annotation_templates** (string category)

Description: Unregisters the template or templates with the given category string on this particular view

See Annotation#unregister_templates for a method doing the same on application level.This method is hardly useful normally, but can be used when customizing layout views as individual widgets.

This method has been added in version 0.28.

unselect_object

Signature: *[const]* void **unselect_object** (const ObjectInstPath obj)

Description: Removes the given selection from the list of selected objects

The selection provided by the ObjectInstPath descriptor is removed from the list of selected objects. If the given object was not part of the selection, nothing will be changed. The selection of other objects (such as annotations and images) will not be affected.

This method has been introduced in version 0.24

update_content

Signature: void **update_content**

Description: Updates the layout view to the current state

This method triggers an update of the hierarchy tree and layer view tree. Usually, this method does not need to be called. The widgets are updated automatically in most cases.

Currently, this method should be called however, after the layer view tree has been changed by the insert_layer, replace_layer_node or delete_layer methods.

viewport_height

Signature: *[const]* int **viewport_height**

Description: Return the viewport height in pixels

This method was introduced in version 0.18.

viewport_trans

Signature: *[const]* DCplxTrans **viewport_trans**

Description: Returns the transformation that converts micron coordinates to pixels

	Hint: the transformation returned will convert any point in micron coordinate space into a pixel coordinate. Contrary to usual convention, the y pixel coordinate is given in a mathematically oriented space - which means the bottom coordinate is 0. This method was introduced in version 0.18.
viewport_width	**Signature**: *[const]* int **viewport_width** **Description**: Returns the viewport width in pixels This method was introduced in version 0.18.
widget	**Signature**: QWidget ptr **widget** **Description**: Gets the QWidget object of the view This method has been introduced in version 0.28.7.
zoom_box	**Signature**: void **zoom_box** (const DBox box) **Description**: Sets the viewport to the given box **box**: The box to which to set the view in micron coordinates
zoom_fit	**Signature**: void **zoom_fit** **Description**: Fits the contents of the current view into the window
zoom_fit_sel	**Signature**: void **zoom_fit_sel** **Description**: Fits the contents of the current selection into the window This method has been introduced in version 0.25.
zoom_in	**Signature**: void **zoom_in** **Description**: Zooms in somewhat
zoom_out	**Signature**: void **zoom_out** **Description**: Zooms out somewhat

4.221. API reference - Class ObjectInstPath

Notation used in Ruby API documentation

Module: lay

Description: A class describing a selected shape or instance

A shape or instance is addressed by a path which describes all instances leading to the specified object. These instances are described through InstElement objects, which specify the instance and, in case of array instances, the specific array member. For shapes, additionally the layer and the shape itself is specified. The ObjectInstPath objects encapsulates both forms, which can be distinguished with the is_cell_inst? predicate.

An instantiation path leads from a top cell down to the container cell which either holds the shape or the instance. The top cell can be obtained through the top attribute, the container cell through the source attribute. Both are cell indexes which can be converted to Cell objects through the Layout#cell. In case of objects located in the top cell, top and source refer to the same cell. The first element of the instantiation path is the instance located within the top cell leading to the first child cell. The second element leads to the next child cell and so forth. path_nth can be used to obtain a specific element of the path.

The cv_index attribute specifies the cellview the selection applies to. Use LayoutView#cellview to obtain the CellView object from the index.

The shape or instance the selection refers to can be obtained with shape and inst respectively. Use is_cell_inst? to decide whether the selection refers to an instance or not.

The ObjectInstPath class plays a role when retrieving and modifying the selection of shapes and instances through LayoutView#object_selection, LayoutView#object_selection=, LayoutView#select_object and LayoutView#unselect_object. ObjectInstPath objects can be modified to reflect a new selection, but the new selection becomes active only after it is installed in the view. The following sample demonstrates that. It implements a function to convert all shapes to polygons:

```
mw = RBA::Application::instance::main_window
view = mw.current_view

begin

  view.transaction("Convert selected shapes to polygons")

  sel = view.object_selection

  sel.each do |s|
    if !s.is_cell_inst? && !s.shape.is_text?
      ly = view.cellview(s.cv_index).layout
      # convert to polygon
      s.shape.polygon = s.shape.polygon
    end
  end

  view.object_selection = sel

ensure
  view.commit
end
```

Note, that without resetting the selection in the above example, the application might raise errors because after modifying the selected objects, the current selection will no longer be valid. Establishing a new valid selection in the way shown above will help avoiding this issue.

Public constructors

new ObjectInstPath ptr	**new**	(const RecursiveShapeIterator si, int cv_index)	Creates a new path object from a RecursiveShapeIterator

Public methods

[const]	bool	**!=**	(const ObjectInstPath b)	Inequality of two ObjectInstPath objects
[const]	bool	**<**	(const ObjectInstPath b)	Provides an order criterion for two ObjectInstPath objects
[const]	bool	**==**	(const ObjectInstPath b)	Equality of two ObjectInstPath objects
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**append_path**	(const InstElement element)	Appends an element to the instantiation path
	void	**assign**	(const ObjectInstPath other)	Assigns another object to self
[const]	unsigned int	**cell_index**		Gets the cell index of the cell that the selection applies to.
	void	**clear_path**		Clears the instantiation path
[const]	unsigned int	**cv_index**		Gets the cellview index that describes which cell view the shape or instance is located in
	void	**cv_index=**	(unsigned int index)	Sets the cellview index that describes which cell view the shape or instance is located in
[const]	DCplxTrans	**dtrans**		Gets the transformation applicable for the shape in micron space.
[const]	new ObjectInstPath ptr	**dup**		Creates a copy of self
[const,iter]	InstElement	**each_inst**		Yields the instantiation path
[const]	Instance	**inst**		Deliver the instance represented by this selection

[const]	bool	**is_cell_inst?**		True, if this selection represents a cell instance
[const]	bool	**is_valid?**	(LayoutView ptr view)	Gets a value indicating whether the instance path refers to a valid object in the context of the given view
[const]	variant	**layer**		Gets the layer index that describes which layer the selected shape is on
	void	**layer=**	(unsigned int layer_index)	Sets to the layer index that describes which layer the selected shape is on
[const]	Layout ptr	**layout**		Gets the Layout object the selected object lives in.
[const]	InstElement[]	**path**		Gets the instantiation path
	void	**path=**	(InstElement[] p)	Sets the instantiation path
[const]	unsigned int	**path_length**		Returns the length of the path (number of elements delivered by each_inst)
[const]	InstElement	**path_nth**	(unsigned int n)	Returns the nth element of the path (similar to each_inst but with direct access through the index)
[const]	unsigned long	**seq**		Gets the sequence number
	void	**seq=**	(unsigned long n)	Sets the sequence number
[const]	variant	**shape**		Gets the selected shape
	void	**shape=**	(const Shape shape)	Sets the shape object that describes the selected shape geometrically
[const]	unsigned int	**source**		Returns to the cell index of the cell that the selected element resides inside.
[const]	DCplxTrans	**source_dtrans**		Gets the transformation applicable for an instance and shape in micron space.
[const]	ICplxTrans	**source_trans**		Gets the transformation applicable for an instance and shape.
[const]	unsigned int	**top**		Gets the cell index of the top cell the selection applies to
	void	**top=**	(unsigned int cell_index)	Sets the cell index of the top cell the selection applies to
[const]	ICplxTrans	**trans**		Gets the transformation applicable for the shape.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead

	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

!=

Signature: *[const]* bool **!=** (const ObjectInstPath b)

Description: Inequality of two ObjectInstPath objects

See the comments on the == operator.

This method has been introduced with version 0.24.

<

Signature: *[const]* bool **<** (const ObjectInstPath b)

Description: Provides an order criterion for two ObjectInstPath objects

Note: this operator is just provided to establish any order, not a particular one.

This method has been introduced with version 0.24.

==

Signature: *[const]* bool **==** (const ObjectInstPath b)

Description: Equality of two ObjectInstPath objects

Note: this operator returns true if both instance paths refer to the same object, not just identical ones.

This method has been introduced with version 0.24.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

append_path

Signature: void **append_path** (const InstElement element)

Description: Appends an element to the instantiation path

This method allows building of an instantiation path pointing to the selected object. For an instance selection, the last component added is the instance which is selected. For a shape selection, the path points to the cell containing the selected shape.

This method was introduced in version 0.24.

assign

Signature: void **assign** (const ObjectInstPath other)

Description: Assigns another object to self

cell_index

Signature: *[const]* unsigned int **cell_index**

Description: Gets the cell index of the cell that the selection applies to.

This method returns the cell index that describes which cell the selected shape is located in or the cell whose instance is selected if is_cell_inst? is true. This property is set implicitly by setting the top cell and adding elements to the instantiation path. To obtain the index of the container cell, use source.

clear_path

Signature: void **clear_path**

Description: Clears the instantiation path

This method was introduced in version 0.24.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

cv_index

Signature: *[const]* unsigned int **cv_index**

Description: Gets the cellview index that describes which cell view the shape or instance is located in

Python specific notes:
The object exposes a readable attribute 'cv_index'. This is the getter.

cv_index=

Signature: void **cv_index=** (unsigned int index)

Description: Sets the cellview index that describes which cell view the shape or instance is located in

This method has been introduced in version 0.24.

Python specific notes:
The object exposes a writable attribute 'cv_index'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dtrans

Signature: *[const]* DCplxTrans **dtrans**

Description: Gets the transformation applicable for the shape in micron space.

This method returns the same transformation than trans, but applicable to objects in micrometer units:

```
# renders the micrometer-unit polygon in top cell coordinates:
dpolygon_in_top = sel.dtrans * sel.shape.dpolygon
```

This method is not applicable to instance selections. A more generic attribute is source_dtrans.

The method has been introduced in version 0.25.

dup

Signature: *[const]* new ObjectInstPath ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_inst

Signature: *[const,iter]* InstElement **each_inst**

Description: Yields the instantiation path

The instantiation path describes by an sequence of InstElement objects the path by which the cell containing the selected shape is found from the cell view's current cell. If this object represents an instance, the path will contain the selected instance as the last element. The elements are delivered top down.

inst

Signature: *[const]* Instance **inst**

Description: Deliver the instance represented by this selection

This method delivers valid results only if is_cell_inst? is true. It returns the instance reference (an Instance object) that this selection represents.

This property is set implicitly by adding instance elements to the instantiation path.

This method has been added in version 0.16.

is_cell_inst?

Signature: *[const]* bool **is_cell_inst?**

Description: True, if this selection represents a cell instance

If this attribute is true, the shape reference and layer are not valid.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_valid?

Signature: *[const]* bool **is_valid?** (LayoutView ptr view)

Description: Gets a value indicating whether the instance path refers to a valid object in the context of the given view

This predicate has been introduced in version 0.27.12.

layer

Signature: *[const]* variant **layer**

Description: Gets the layer index that describes which layer the selected shape is on

Starting with version 0.27, this method returns nil for this property if is_cell_inst? is false - i.e. the selection does not represent a shape.

Python specific notes:
The object exposes a readable attribute 'layer'. This is the getter.

layer=

Signature: void **layer=** (unsigned int layer_index)

Description: Sets to the layer index that describes which layer the selected shape is on

Setting the layer property to a valid layer index makes the path a shape selection path. Setting the layer property to a negative layer index makes the selection an instance selection.

This method has been introduced in version 0.24.

Python specific notes:
The object exposes a writable attribute 'layer'. This is the setter.

layout

Signature: *[const]* Layout ptr **layout**

Description: Gets the Layout object the selected object lives in.

This method returns the Layout object that the selected object lives in. This method may return nil, if the selection does not point to a valid object.

This method has been introduced in version 0.25.

new

Signature: *[static]* new ObjectInstPath ptr **new** (const RecursiveShapeIterator si, int cv_index)

Description: Creates a new path object from a RecursiveShapeIterator

Use this constructor to quickly turn a recursive shape iterator delivery into a shape selection.

Python specific notes:
This method is the default initializer of the object.

path

Signature: *[const]* InstElement[] **path**

Description: Gets the instantiation path

The path is a sequence of InstElement objects leading to the target object.

This method was introduced in version 0.26.

Python specific notes:
The object exposes a readable attribute 'path'. This is the getter.

path=

Signature: void **path=** (InstElement[] p)

Description: Sets the instantiation path

This method was introduced in version 0.26.

Python specific notes:
The object exposes a writable attribute 'path'. This is the setter.

path_length

Signature: *[const]* unsigned int **path_length**

Description: Returns the length of the path (number of elements delivered by each_inst)

This method has been added in version 0.16.

path_nth

Signature: *[const]* InstElement **path_nth** (unsigned int n)

Description: Returns the nth element of the path (similar to each_inst but with direct access through the index)

n: The index of the element to retrieve (0..path_length-1)

This method has been added in version 0.16.

seq

Signature: *[const]* unsigned long **seq**

Description: Gets the sequence number

The sequence number describes when the item was selected. A sequence number of 0 indicates that the item was selected in the first selection action (without 'Shift' pressed).

Python specific notes:
The object exposes a readable attribute 'seq'. This is the getter.

seq=

Signature: void **seq=** (unsigned long n)

Description: Sets the sequence number

See seq for a description of this property.

This method was introduced in version 0.24.

Python specific notes:
The object exposes a writable attribute 'seq'. This is the setter.

shape

Signature: *[const]* variant **shape**

Description: Gets the selected shape

The shape object may be modified. This does not have an immediate effect on the selection. Instead, the selection must be set in the view using LayoutView#object_selection= or LayoutView#select_object.

This method delivers valid results only for object selections that represent shapes. Starting with version 0.27, this method returns nil for this property if is_cell_inst? is false.

Python specific notes:
The object exposes a readable attribute 'shape'. This is the getter.

shape=

Signature: void **shape=** (const Shape shape)

Description: Sets the shape object that describes the selected shape geometrically

When using this setter, the layer index must be set to a valid layout layer (see layer=). Setting both properties makes the selection a shape selection.

This method has been introduced in version 0.24.

Python specific notes:
The object exposes a writable attribute 'shape'. This is the setter.

source

Signature: *[const]* unsigned int **source**

Description: Returns to the cell index of the cell that the selected element resides inside.

If this reference represents a cell instance, this method delivers the index of the cell in which the cell instance resides. Otherwise, this method returns the same value than cell_index.

This property is set implicitly by setting the top cell and adding elements to the instantiation path.

This method has been added in version 0.16.

source_dtrans

Signature: *[const]* DCplxTrans **source_dtrans**

Description: Gets the transformation applicable for an instance and shape in micron space.

This method returns the same transformation than source_trans, but applicable to objects in micrometer units:

```
# renders the cell instance as seen from top level:
dcell_inst_in_top = sel.source_dtrans * sel.inst.dcell_inst
```

The method has been introduced in version 0.25.

source_trans

Signature: *[const]* ICplxTrans **source_trans**

Description: Gets the transformation applicable for an instance and shape.

If this object represents a shape, this transformation describes how the selected shape is transformed into the current cell of the cell view. If this object represents an instance, this transformation describes how the selected instance is transformed into the current cell of the cell view. This method is similar to trans, except that the resulting transformation does not include the instance transformation if the object represents an instance.

This property is set implicitly by setting the top cell and adding elements to the instantiation path.

This method has been added in version 0.16.

top

Signature: *[const]* unsigned int **top**

Description: Gets the cell index of the top cell the selection applies to

	The top cell is identical to the current cell provided by the cell view. It is the cell from which is instantiation path originates and the container cell if not instantiation path is set. This method has been introduced in version 0.24. **Python specific notes:** The object exposes a readable attribute 'top'. This is the getter.
top=	**Signature**: void **top=** (unsigned int cell_index) **Description**: Sets the cell index of the top cell the selection applies to See top_cell for a description of this property. This method has been introduced in version 0.24. **Python specific notes:** The object exposes a writable attribute 'top'. This is the setter.
trans	**Signature**: *[const]* ICplxTrans **trans** **Description**: Gets the transformation applicable for the shape. If this object represents a shape, this transformation describes how the selected shape is transformed into the current cell of the cell view. Basically, this transformation is the accumulated transformation over the instantiation path. If the ObjectInstPath represents a cell instance, this includes the transformation of the selected instance as well. This property is set implicitly by setting the top cell and adding elements to the instantiation path. This method is not applicable for instance selections. A more generic attribute is source_trans.

4.222. API reference - Class MacroExecutionContext

Notation used in Ruby API documentation

Module: lay

Description: Support for various debugger features

This class implements some features that allow customization of the debugger behavior, specifically the generation of back traces and the handling of exception. These functions are particular useful for implementing DSL interpreters and providing proper error locations in the back traces or to suppress exceptions when re-raising them.

Public constructors

new MacroExecutionContext ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const MacroExec other)	Assigns another object to self
[const]	new MacroExecutionContext ptr	**dup**		Creates a copy of self

Public static methods and constants

void	**ignore_next_exception**		Ignores the next exception in the debugger
void	**remove_debugger_scope**		Removes a debugger scope previously set with set_debugger_scope
void	**set_debugger_scope**	(string filename)	Sets a debugger scope (file level which shall appear in the debugger)

Deprecated methods (protected, public, static, non-static and constructors)

void	**create**		Use of this method is deprecated. Use _create instead
void	**destroy**		Use of this method is deprecated. Use _destroy instead

[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const MacroExecutionContext other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new MacroExecutionContext ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

ignore_next_exception

Signature: *[static]* void **ignore_next_exception**

Description: Ignores the next exception in the debugger

The next exception thrown will be ignored in the debugger. That feature is useful when re-raising exceptions if those new exception shall not appear in the debugger.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new MacroExecutionContext ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

remove_debugger_scope

Signature: *[static]* void **remove_debugger_scope**

Description: Removes a debugger scope previously set with set_debugger_scope

set_debugger_scope

Signature: *[static]* void **set_debugger_scope** (string filename)

Description: Sets a debugger scope (file level which shall appear in the debugger)

If a debugger scope is set, back traces will be produced starting from that scope. Setting a scope is useful for implementing DSL interpreters and giving a proper hint about the original location of an error.

4.223. API reference - Class MacroInterpreter

Notation used in Ruby API documentation

Module: lay

Description: A custom interpreter for a DSL (domain specific language)

DSL interpreters are a way to provide macros written in a language specific for the application. One example are DRC scripts which are written in some special language optimized for DRC ruledecks. Interpreters for such languages can be built using scripts itself by providing the interpreter implementation through this object.

An interpreter implementation involves at least these steps:

- Derive a new object from RBA::MacroInterpreter
- Reimplement the execute method for the actual execution of the code
- In the initialize method configure the object using the attribute setters like suffix= and register the object as DSL interpreter (in that order)
- Create at least one template macro in the initialize method

Template macros provide a way for the macro editor to present macros for the new interpreter in the list of templates. Template macros can provide menu bindings, shortcuts and some initial text for example

The simple implementation can be enhanced by providing more information, i.e. syntax highlighter information, the debugger to use etc. This involves reimplementing further methods, i.e. "syntax_scheme".

This is a simple example for an interpreter in Ruby. Is is registered under the name 'simple-dsl' and just evaluates the script text:

```
class SimpleExecutable < RBA::Excutable

  # Constructor
  def initialize(macro)
    @macro = macro
  end

  # Implements the execute method
  def execute
    eval(@macro.text, nil, @macro.path)
    nil
  end

end

class SimpleInterpreter < RBA::MacroInterpreter

  # Constructor
  def initialize
    self.description = "A test interpreter"
    # Registers the new interpreter
    register("simple-dsl")
    # create a template for the macro editor:
    # Name is "new_simple", the description will be "Simple interpreter macro"
    # in the "Special" group.
    mt = create_template("new_simple")
    mt.description = "Special;;Simple interpreter macro"
  end

  # Creates the executable delegate
  def executable(macro)
    SimpleExecutable::new(macro)
  end

end
```

```
# Register the new interpreter
SimpleInterpreter::new
```

Please note that such an implementation is dangerous because the evaluation of the script happens in the context of the interpreter object. In this implementation the script could redefine the execute method for example. This implementation is provided as an example only. A real implementation should add execution of prolog and epilog code inside the execute method and proper error handling.

In order to make the above code effective, store the code in an macro, set "early auto-run" and restart KLayout.

This class has been introduced in version 0.23 and modified in 0.27.

Public constructors

new MacroInterpreter ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const MacroInterpreter other)	Assigns another object to self
	Macro ptr	**create_template**	(string url)	Creates a new macro template
	void	**debugger_scheme=**	(Macro::Interpreter scheme)	Sets the debugger scheme (which debugger to use for the DSL macro)
	void	**description=**	(string description)	Sets a description string
[const]	new MacroInterpreter ptr	**dup**		Creates a copy of self
[virtual,const]	Executable ptr	**executable**	(const Macro ptr macro)	Returns the executable object which implements the macro execution
	string[]	**include_expansion**	(Macro ptr macro)	Provides include expansion as defined by the interpreter
	void	**register**	(string name)	Registers the macro interpreter

void	storage_scheme=	(Macro::Format scheme)	Sets the storage scheme (the format as which the macro is stored)
void	suffix=	(string suffix)	Sets the file suffix
void	supports_include_expa	(bool flag)	Sets a value indicating whether this interpreter supports the default include file expansion scheme.
void	syntax_scheme=	(string scheme)	Sets a string indicating the syntax highlighter scheme

Public static methods and constants

[static,const]	Macro::Format	MacroFormat	The macro has macro (XML) format
	Macro::Interpreter	NoDebugger	Indicates no debugging for debugger_scheme
[static,const]	Macro::Format	PlainTextFormat	The macro has plain text format
[static,const]	Macro::Format	PlainTextWithHashAnnotationsFormat	The macro has plain text format with special pseudo-comment annotations
	Macro::Interpreter	RubyDebugger	Indicates Ruby debugger for debugger_scheme

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

MacroFormat

Signature: *[static,const]* Macro::Format **MacroFormat**

Description: The macro has macro (XML) format

Python specific notes:
The object exposes a readable attribute 'MacroFormat'. This is the getter.

NoDebugger

Signature: *[static]* Macro::Interpreter **NoDebugger**

Description: Indicates no debugging for debugger_scheme

Python specific notes:
The object exposes a readable attribute 'NoDebugger'. This is the getter.

PlainTextFormat

Signature: *[static,const]* Macro::Format **PlainTextFormat**

Description: The macro has plain text format

Python specific notes:
The object exposes a readable attribute 'PlainTextFormat'. This is the getter.

PlainTextWithHashAnnotationsFormat

Signature: *[static,const]* Macro::Format **PlainTextWithHashAnnotationsFormat**

Description: The macro has plain text format with special pseudo-comment annotations

Python specific notes:
The object exposes a readable attribute 'PlainTextWithHashAnnotationsFormat'. This is the getter.

RubyDebugger

Signature: *[static]* Macro::Interpreter **RubyDebugger**

Description: Indicates Ruby debugger for debugger_scheme

Python specific notes:
The object exposes a readable attribute 'RubyDebugger'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const MacroInterpreter other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

create_template

Signature: Macro ptr **create_template** (string url)

Description: Creates a new macro template

url: The template will be initialized from that URL.

This method will create a register a new macro template. It returns a Macro object which can be modified in order to adjust the template (for example to set description, add a content, menu binding, autorun flags etc.)

This method must be called after register has called.

debugger_scheme=

Signature: void **debugger_scheme=** (Macro::Interpreter scheme)

Description: Sets the debugger scheme (which debugger to use for the DSL macro)

The value can be one of the constants RubyDebugger or NoDebugger.

Use this attribute setter in the initializer before registering the interpreter.

Before version 0.25 this attribute was a re-implementable method. It has been turned into an attribute for performance reasons in version 0.25.

Python specific notes:
The object exposes a writable attribute 'debugger_scheme'. This is the setter.

description=

Signature: void **description=** (string description)

Description: Sets a description string

This string is used for showing the type of DSL macro in the file selection box together with the suffix for example. Use this attribute setter in the initializer before registering the interpreter.

Before version 0.25 this attribute was a re-implementable method. It has been turned into an attribute for performance reasons in version 0.25.

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new MacroInterpreter ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

executable

Signature: *[virtual,const]* Executable ptr **executable** (const Macro ptr macro)

Description: Returns the executable object which implements the macro execution

macro: The macro to execute

This method must be reimplemented to return an Executable object for the actual implementation. The system will use this function to execute the script when a macro with interpreter type 'dsl' and the name of this interpreter is run.

This method has been introduced in version 0.27 and replaces the 'execute' method.

include_expansion

Signature: string[] **include_expansion** (Macro ptr macro)

Description: Provides include expansion as defined by the interpreter

The return value will be a two-element array with the encoded file path and the include-expanded text.

This method has been introduced in version 0.28.12.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new MacroInterpreter ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

register

Signature: void **register** (string name)

Description: Registers the macro interpreter

name: The interpreter name. This is an arbitrary string which should be unique.

Registration of the interpreter makes the object known to the system. After registration, macros whose interpreter is set to 'dsl' can use this object to run the script. For executing a script, the system will call the interpreter's execute method.

storage_scheme=

Signature: void **storage_scheme=** (Macro::Format scheme)

Description: Sets the storage scheme (the format as which the macro is stored)

This value indicates how files for this DSL macro type shall be stored. The value can be one of the constants PlainTextFormat, PlainTextWithHashAnnotationsFormat and MacroFormat.

Use this attribute setter in the initializer before registering the interpreter.

Before version 0.25 this attribute was a re-implementable method. It has been turned into an attribute for performance reasons in version 0.25.

Python specific notes:
The object exposes a writable attribute 'storage_scheme'. This is the setter.

suffix=

Signature: void **suffix=** (string suffix)

Description: Sets the file suffix

This string defines which file suffix to associate with the DSL macro. If an empty string is given (the default) no particular suffix is assciated with that macro type and "lym" is assumed. Use this attribute setter in the initializer before registering the interpreter.

Before version 0.25 this attribute was a re-implementable method. It has been turned into an attribute for performance reasons in version 0.25.

Python specific notes:
The object exposes a writable attribute 'suffix'. This is the setter.

supports_include_expansio

Signature: void **supports_include_expansion=** (bool flag)

Description: Sets a value indicating whether this interpreter supports the default include file expansion scheme.

If this value is set to true (the default), lines like '# %include ...' will be substituted by the content of the file following the '%include' keyword. Set this value to false if you don't want to support this feature.

This attribute has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'supports_include_expansion'. This is the setter.

syntax_scheme=

Signature: void **syntax_scheme=** (string scheme)

Description: Sets a string indicating the syntax highlighter scheme

The scheme string can be empty (indicating no syntax highlighting), "ruby" for the Ruby syntax highlighter or another string. In that case, the highlighter will look for a syntax definition under the resource path ":/syntax/<scheme>.xml".

Use this attribute setter in the initializer before registering the interpreter.

Before version 0.25 this attribute was a re-implementable method. It has been turned into an attribute for performance reasons in version 0.25.

Python specific notes:
The object exposes a writable attribute 'syntax_scheme'. This is the setter.

4.224. API reference - Class Macro

Notation used in Ruby API documentation

Module: lay

Description: A macro class

Sub-classes: Format, Interpreter

This class is provided mainly to support generation of template macros in the DSL interpreter framework provided by MacroInterpreter. The implementation may be enhanced in future versions and provide access to macros stored inside KLayout's macro repository. But it can be used to execute macro code in a consistent way:

```
path = "path-to-macro.lym"
RBA::Macro::new(path).run()
```

Using the Macro class with run for executing code will chose the right interpreter and is able to execute DRC and LVS scripts in the proper environment. This also provides an option to execute Ruby code from Python and vice versa.

In this scenario you can pass values to the script using Interpreter#define_variable. The interpreter to choose for DRC and LVS scripts is Interpreter#ruby_interpreter. For passing values back from the script, wrap the variable value into a Value object which can be modified by the called script and read back by the caller.

Public constructors

new Macro ptr	**new**	(string path)		Loads the macro from the given file path

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
[const]	string	**category**		Gets the category tags
	void	**category=**	(string string)	Sets the category tags string
[const]	string	**description**		Gets the description text
	void	**description=**	(string description)	Sets the description text
[const]	string	**doc**		Gets the macro's documentation string

	void	**doc=**	(string doc)	Sets the macro's documentation string
[const]	string	**dsl_interpreter**		Gets the macro's DSL interpreter name (if interpreter is DSLInterpreter)
	void	**dsl_interpreter=**	(string dsl_interpreter)	Sets the macro's DSL interpreter name (if interpreter is DSLInterpreter)
[const]	string	**epilog**		Gets the epilog code
	void	**epilog=**	(string string)	Sets the epilog
[const]	Macro::Format	**format**		Gets the macro's storage format
	void	**format=**	(Macro::Format format)	Sets the macro's storage format
[const]	string	**group_name**		Gets the menu group name
	void	**group_name=**	(string string)	Sets the menu group name
[const]	Macro::Interpreter	**interpreter**		Gets the macro's interpreter
	void	**interpreter=**	(Macro::Interpreter interpreter)	Sets the macro's interpreter
[const]	string	**interpreter_name**		Gets the macro interpreter name
	void	**is_autorun=**	(bool flag)	Sets a flag indicating whether the macro is automatically executed on startup
[const]	bool	**is_autorun?**		Gets a flag indicating whether the macro is automatically executed on startup
	void	**is_autorun_early=**	(bool flag)	Sets a flag indicating whether the macro is automatically executed early on startup
[const]	bool	**is_autorun_early?**		Gets a flag indicating whether the macro is automatically executed early on startup
[const]	string	**menu_path**		Gets the menu path
	void	**menu_path=**	(string string)	Sets the menu path
[const]	string	**name**		Gets the name of the macro
[const]	string	**path**		Gets the path of the macro
[const]	string	**prolog**		Gets the prolog code
	void	**prolog=**	(string string)	Sets the prolog
[const]	int	**run**		Executes the macro
	void	**save_to**	(string path)	Saves the macro to the given file
[const]	string	**shortcut**		Gets the macro's keyboard shortcut

	void	**shortcut=**	(string shortcut)	Sets the macro's keyboard shortcut
	void	**show_in_menu=**	(bool flag)	Sets a value indicating whether the macro shall be shown in the menu
[const]	bool	**show_in_menu?**		Gets a value indicating whether the macro shall be shown in the menu
	void	**sync_properties_with_text**		Synchronizes the macro properties with the text
	void	**sync_text_with_propertie**		Synchronizes the macro text with the properties
[const]	string	**text**		Gets the macro text
	void	**text=**	(string string)	Sets the macro text
[const]	string	**version**		Gets the macro's version
	void	**version=**	(string version)	Sets the macro's version

Public static methods and constants

[static,const]	Macro::Interpreter	**DSLInterpreter**		A domain-specific interpreter (DSL)
[static,const]	Macro::Format	**MacroFormat**		The macro has macro (XML) format
[static,const]	Macro::Interpreter	**None**		No specific interpreter
[static,const]	Macro::Format	**PlainTextFormat**		The macro has plain text format
[static,const]	Macro::Format	**PlainTextWithHashAnn**		The macro has plain text format with special pseudo-comment annotations
[static,const]	Macro::Interpreter	**Python**		The interpreter is Python
[static,const]	Macro::Interpreter	**Ruby**		The interpreter is Ruby
[static,const]	Macro::Interpreter	**Text**		Plain text
	Macro ptr	**macro_by_path**	(string path)	Finds the macro by installation path
	int	**real_line**	(string path, int line)	Gets the real line number for an include-encoded path and line number
	string	**real_path**	(string path, int line)	Gets the real path for an include-encoded path and line number

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

DSLInterpreter

Signature: *[static,const]* Macro::Interpreter **DSLInterpreter**

Description: A domain-specific interpreter (DSL)

Python specific notes:
The object exposes a readable attribute 'DSLInterpreter'. This is the getter.

MacroFormat

Signature: *[static,const]* Macro::Format **MacroFormat**

Description: The macro has macro (XML) format

Python specific notes:
The object exposes a readable attribute 'MacroFormat'. This is the getter.

None

Signature: *[static,const]* Macro::Interpreter **None**

Description: No specific interpreter

Python specific notes:
This member is available as 'None_' in Python.
The object exposes a readable attribute 'None_'. This is the getter.

PlainTextFormat

Signature: *[static,const]* Macro::Format **PlainTextFormat**

Description: The macro has plain text format

Python specific notes:
The object exposes a readable attribute 'PlainTextFormat'. This is the getter.

PlainTextWithHashAnnotat

Signature: *[static,const]* Macro::Format **PlainTextWithHashAnnotationsFormat**

Description: The macro has plain text format with special pseudo-comment annotations

Python specific notes:
The object exposes a readable attribute 'PlainTextWithHashAnnotationsFormat'. This is the getter.

Python

Signature: *[static,const]* Macro::Interpreter **Python**

Description: The interpreter is Python

Python specific notes:
The object exposes a readable attribute 'Python'. This is the getter.

Ruby

Signature: *[static,const]* Macro::Interpreter **Ruby**

Description: The interpreter is Ruby

Python specific notes:

The object exposes a readable attribute 'Ruby'. This is the getter.

Text

Signature: *[static,const]* Macro::Interpreter **Text**

Description: Plain text

Python specific notes:
The object exposes a readable attribute 'Text'. This is the getter.

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

category

Signature: *[const]* string **category**

Description: Gets the category tags

The category tags string indicates to which categories a macro will belong to. This string is only used for templates currently and is a comma-separated list of category names.

Python specific notes:
The object exposes a readable attribute 'category'. This is the getter.

category=

Signature: void **category=** (string string)

Description: Sets the category tags string

See category for details.

Python specific notes:
The object exposes a writable attribute 'category'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

description

Signature: *[const]* string **description**

Description: Gets the description text

The description text of a macro will appear in the macro list. If used as a macro template, the description text can have the format "Group;;Description". In that case, the macro will appear in a group with title "Group".

Python specific notes:
The object exposes a readable attribute 'description'. This is the getter.

description=

Signature: void **description=** (string description)

Description: Sets the description text

description: The description text.

See description for details.

Python specific notes:
The object exposes a writable attribute 'description'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

	This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.
doc	**Signature**: *[const]* string **doc** **Description**: Gets the macro's documentation string This method has been introduced in version 0.27.5. **Python specific notes:** The object exposes a readable attribute 'doc'. This is the getter.
doc=	**Signature**: void **doc=** (string doc) **Description**: Sets the macro's documentation string This method has been introduced in version 0.27.5. **Python specific notes:** The object exposes a writable attribute 'doc'. This is the setter.
dsl_interpreter	**Signature**: *[const]* string **dsl_interpreter** **Description**: Gets the macro's DSL interpreter name (if interpreter is DSLInterpreter) This method has been introduced in version 0.27.5. **Python specific notes:** The object exposes a readable attribute 'dsl_interpreter'. This is the getter.
dsl_interpreter=	**Signature**: void **dsl_interpreter=** (string dsl_interpreter) **Description**: Sets the macro's DSL interpreter name (if interpreter is DSLInterpreter) This method has been introduced in version 0.27.5. **Python specific notes:** The object exposes a writable attribute 'dsl_interpreter'. This is the setter.
epilog	**Signature**: *[const]* string **epilog** **Description**: Gets the epilog code The epilog is executed after the actual code is executed. Interpretation depends on the implementation of the DSL interpreter for DSL macros. **Python specific notes:** The object exposes a readable attribute 'epilog'. This is the getter.
epilog=	**Signature**: void **epilog=** (string string) **Description**: Sets the epilog See epilog for details. **Python specific notes:** The object exposes a writable attribute 'epilog'. This is the setter.
format	**Signature**: *[const]* Macro::Format **format** **Description**: Gets the macro's storage format This method has been introduced in version 0.27.5. **Python specific notes:**

The object exposes a readable attribute 'format'. This is the getter.

format=

Signature: void **format=** (Macro::Format format)

Description: Sets the macro's storage format

This method has been introduced in version 0.27.5.

Python specific notes:
The object exposes a writable attribute 'format'. This is the setter.

group_name

Signature: *[const]* string **group_name**

Description: Gets the menu group name

If a group name is specified and show_in_menu? is true, the macro will appear in a separate group (separated by a separator) together with other macros sharing the same group.

Python specific notes:
The object exposes a readable attribute 'group_name'. This is the getter.

group_name=

Signature: void **group_name=** (string string)

Description: Sets the menu group name

See group_name for details.

Python specific notes:
The object exposes a writable attribute 'group_name'. This is the setter.

interpreter

Signature: *[const]* Macro::Interpreter **interpreter**

Description: Gets the macro's interpreter

This method has been introduced in version 0.27.5.

Python specific notes:
The object exposes a readable attribute 'interpreter'. This is the getter.

interpreter=

Signature: void **interpreter=** (Macro::Interpreter interpreter)

Description: Sets the macro's interpreter

This method has been introduced in version 0.27.5.

Python specific notes:
The object exposes a writable attribute 'interpreter'. This is the setter.

interpreter_name

Signature: *[const]* string **interpreter_name**

Description: Gets the macro interpreter name

This is the string version of interpreter.

This method has been introduced in version 0.27.5.

is_autorun=

Signature: void **is_autorun=** (bool flag)

Description: Sets a flag indicating whether the macro is automatically executed on startup

This method has been introduced in version 0.27.5.

Python specific notes:
The object exposes a writable attribute 'is_autorun'. This is the setter.

is_autorun?

Signature: *[const]* bool **is_autorun?**

Description: Gets a flag indicating whether the macro is automatically executed on startup

This method has been introduced in version 0.27.5.

Python specific notes:
The object exposes a readable attribute 'is_autorun'. This is the getter.

is_autorun_early=

Signature: void **is_autorun_early=** (bool flag)

Description: Sets a flag indicating whether the macro is automatically executed early on startup

This method has been introduced in version 0.27.5.

Python specific notes:
The object exposes a writable attribute 'is_autorun_early'. This is the setter.

is_autorun_early?

Signature: *[const]* bool **is_autorun_early?**

Description: Gets a flag indicating whether the macro is automatically executed early on startup

This method has been introduced in version 0.27.5.

Python specific notes:
The object exposes a readable attribute 'is_autorun_early'. This is the getter.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

macro_by_path

Signature: *[static]* Macro ptr **macro_by_path** (string path)

Description: Finds the macro by installation path

Returns nil if no macro with this path can be found.

This method has been added in version 0.26.

menu_path

Signature: *[const]* string **menu_path**

Description: Gets the menu path

If a menu path is specified and show_in_menu? is true, the macro will appear in the menu at the specified position.

Python specific notes:
The object exposes a readable attribute 'menu_path'. This is the getter.

menu_path=

Signature: void **menu_path=** (string string)

Description: Sets the menu path

See menu_path for details.

Python specific notes:
The object exposes a writable attribute 'menu_path'. This is the setter.

name

Signature: *[const]* string **name**

Description: Gets the name of the macro

This attribute has been added in version 0.25.

new

Signature: *[static]* new Macro ptr **new** (string path)

Description: Loads the macro from the given file path

This constructor has been introduced in version 0.27.5.

Python specific notes:
This method is the default initializer of the object.

path

Signature: *[const]* string **path**

Description: Gets the path of the macro

The path is the path where the macro is stored, starting with an abstract group identifier. The path is used to identify the macro in the debugger for example.

prolog

Signature: *[const]* string **prolog**

Description: Gets the prolog code

The prolog is executed before the actual code is executed. Interpretation depends on the implementation of the DSL interpreter for DSL macros.

Python specific notes:
The object exposes a readable attribute 'prolog'. This is the getter.

prolog=

Signature: void **prolog=** (string string)

Description: Sets the prolog

See prolog for details.

Python specific notes:
The object exposes a writable attribute 'prolog'. This is the setter.

real_line

Signature: *[static]* int **real_line** (string path, int line)

Description: Gets the real line number for an include-encoded path and line number

When using KLayout's include scheme based on '# %include ...', __FILE__ and __LINE__ (Ruby) will not have the proper values but encoded file names. This method allows retrieving the real line number by using

```
# Ruby
real_line = RBA::Macro::real_line(__FILE__, __LINE__)

# Python
real_line = pya::Macro::real_line(__file__, __line__)
```

This substitution is not required for top-level macros as KLayout's interpreter will automatically use this function instead of __FILE__. Call this function when you need __FILE__ from files included through the languages mechanisms such as 'require' or 'load' where this substitution does not happen.

For Python there is no equivalent for __LINE__, so you always have to use:

```
# Pythonimport inspect
```

```
real_line = pya.Macro.real_line(__file__,
 inspect.currentframe().f_back.f_lineno)
```

This feature has been introduced in version 0.27.

real_path

Signature: *[static]* string **real_path** (string path, int line)

Description: Gets the real path for an include-encoded path and line number

When using KLayout's include scheme based on '# %include ...', __FILE__ and __LINE__ (Ruby) will not have the proper values but encoded file names. This method allows retrieving the real file by using

```
# Ruby
real_file = RBA::Macro::real_path(__FILE__, __LINE__)
```

This substitution is not required for top-level macros as KLayout's interpreter will automatically use this function instead of __FILE__. Call this function when you need __FILE__ from files included through the languages mechanisms such as 'require' or 'load' where this substitution does not happen.

For Python there is no equivalent for __LINE__, so you always have to use:

```
# Pythonimport inspect
real_file = pya.Macro.real_path(__file__,
 inspect.currentframe().f_back.f_lineno)
```

This feature has been introduced in version 0.27.

run

Signature: *[const]* int **run**

Description: Executes the macro

This method has been introduced in version 0.27.5.

save_to

Signature: void **save_to** (string path)

Description: Saves the macro to the given file

This method has been introduced in version 0.27.5.

shortcut

Signature: *[const]* string **shortcut**

Description: Gets the macro's keyboard shortcut

This method has been introduced in version 0.27.5.

Python specific notes:
The object exposes a readable attribute 'shortcut'. This is the getter.

shortcut=

Signature: void **shortcut=** (string shortcut)

Description: Sets the macro's keyboard shortcut

This method has been introduced in version 0.27.5.

Python specific notes:

	The object exposes a writable attribute 'shortcut'. This is the setter.
show_in_menu=	**Signature**: void **show_in_menu=** (bool flag) **Description**: Sets a value indicating whether the macro shall be shown in the menu **Python specific notes:** The object exposes a writable attribute 'show_in_menu'. This is the setter.
show_in_menu?	**Signature**: *[const]* bool **show_in_menu?** **Description**: Gets a value indicating whether the macro shall be shown in the menu **Python specific notes:** The object exposes a readable attribute 'show_in_menu'. This is the getter.
sync_properties_with_text	**Signature**: void **sync_properties_with_text** **Description**: Synchronizes the macro properties with the text This method performs the reverse process of sync_text_with_properties. This method has been introduced in version 0.27.5.
sync_text_with_properties	**Signature**: void **sync_text_with_properties** **Description**: Synchronizes the macro text with the properties This method applies to PlainTextWithHashAnnotationsFormat format. The macro text will be enhanced with pseudo-comments reflecting the macro properties. This way, the macro properties can be stored in plain files. This method has been introduced in version 0.27.5.
text	**Signature**: *[const]* string **text** **Description**: Gets the macro text The text is the code executed by the macro interpreter. Depending on the DSL interpreter, the text can be any kind of code. **Python specific notes:** The object exposes a readable attribute 'text'. This is the getter.
text=	**Signature**: void **text=** (string string) **Description**: Sets the macro text See text for details. **Python specific notes:** The object exposes a writable attribute 'text'. This is the setter.
version	**Signature**: *[const]* string **version** **Description**: Gets the macro's version This method has been introduced in version 0.27.5. **Python specific notes:** The object exposes a readable attribute 'version'. This is the getter.
version=	**Signature**: void **version=** (string version) **Description**: Sets the macro's version

This method has been introduced in version 0.27.5.

Python specific notes:
The object exposes a writable attribute 'version'. This is the setter.

4.225. API reference - Class Macro::Format

Notation used in Ruby API documentation

Module: lay

Description: Specifies the format of a macro

This class is equivalent to the class Macro::Format

This enum has been introduced in version 0.27.5.

Public constructors

new Macro::Format ptr	**new**	(int i)	Creates an enum from an integer value
new Macro::Format ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const Macro::Format other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const Macro::Format other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const Macro::Format other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	Macro::Format	**MacroFormat**	The macro has macro (XML) format
[static,const]	Macro::Format	**PlainTextFormat**	The macro has plain text format
[static,const]	Macro::Format	**PlainTextWithHashAnnotat**	The macro has plain text format with special pseudo-comment annotations

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const Macro::Format other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const Macro::Format other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const Macro::Format other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
MacroFormat	**Signature**: *[static,const]* Macro::Format **MacroFormat** **Description**: The macro has macro (XML) format **Python specific notes:** The object exposes a readable attribute 'MacroFormat'. This is the getter.
PlainTextFormat	**Signature**: *[static,const]* Macro::Format **PlainTextFormat** **Description**: The macro has plain text format **Python specific notes:** The object exposes a readable attribute 'PlainTextFormat'. This is the getter.
PlainTextWithHashAnnotationsFormat	**Signature**: *[static,const]* Macro::Format **PlainTextWithHashAnnotationsFormat** **Description**: The macro has plain text format with special pseudo-comment annotations **Python specific notes:** The object exposes a readable attribute 'PlainTextWithHashAnnotationsFormat'. This is the getter.
hash	**Signature**: *[const]* int **hash** **Description**: Gets the hash value from the enum **Python specific notes:** This method is also available as 'hash(object)'.
inspect	**Signature**: *[const]* string **inspect** **Description**: Converts an enum to a visual string **Python specific notes:** This method is also available as 'repr(object)'.

new	**(1) Signature**: *[static]* new Macro::Format ptr **new** (int i) **Description**: Creates an enum from an integer value **Python specific notes:** This method is the default initializer of the object. **(2) Signature**: *[static]* new Macro::Format ptr **new** (string s) **Description**: Creates an enum from a string value **Python specific notes:** This method is the default initializer of the object.
to_i	**Signature**: *[const]* int **to_i** **Description**: Gets the integer value from the enum **Python specific notes:** This method is also available as 'int(object)'.
to_s	**Signature**: *[const]* string **to_s** **Description**: Gets the symbolic string from an enum **Python specific notes:** This method is also available as 'str(object)'.

4.226. API reference - Class Macro::Interpreter

Notation used in Ruby API documentation

Module: lay

Description: Specifies the interpreter used for executing a macro

This class is equivalent to the class Macro::Interpreter

This enum has been introduced in version 0.27.5.

Public constructors

new Macro::Interpreter ptr	**new**	(int i)	Creates an enum from an integer value
new Macro::Interpreter ptr	**new**	(string s)	Creates an enum from a string value

Public methods

[const]	bool	**!=**	(const Macro::Interpreter other)	Compares two enums for inequality
[const]	bool	**!=**	(int other)	Compares an enum with an integer for inequality
[const]	bool	**<**	(const Macro::Interpreter other)	Returns true if the first enum is less (in the enum symbol order) than the second
[const]	bool	**<**	(int other)	Returns true if the enum is less (in the enum symbol order) than the integer value
[const]	bool	**==**	(const Macro::Interpreter other)	Compares two enums
[const]	bool	**==**	(int other)	Compares an enum with an integer value
[const]	int	**hash**		Gets the hash value from the enum
[const]	string	**inspect**		Converts an enum to a visual string
[const]	int	**to_i**		Gets the integer value from the enum
[const]	string	**to_s**		Gets the symbolic string from an enum

Public static methods and constants

[static,const]	Macro::Interpreter	**DSLInterpreter**	A domain-specific interpreter (DSL)
[static,const]	Macro::Interpreter	**None**	No specific interpreter
[static,const]	Macro::Interpreter	**Python**	The interpreter is Python
[static,const]	Macro::Interpreter	**Ruby**	The interpreter is Ruby

[static,const]	Macro::Interpreter	**Text**	Plain text

Detailed description

!=	**(1) Signature**: *[const]* bool **!=** (const Macro::Interpreter other) **Description**: Compares two enums for inequality **(2) Signature**: *[const]* bool **!=** (int other) **Description**: Compares an enum with an integer for inequality
<	**(1) Signature**: *[const]* bool **<** (const Macro::Interpreter other) **Description**: Returns true if the first enum is less (in the enum symbol order) than the second **(2) Signature**: *[const]* bool **<** (int other) **Description**: Returns true if the enum is less (in the enum symbol order) than the integer value
==	**(1) Signature**: *[const]* bool **==** (const Macro::Interpreter other) **Description**: Compares two enums **(2) Signature**: *[const]* bool **==** (int other) **Description**: Compares an enum with an integer value
DSLInterpreter	**Signature**: *[static,const]* Macro::Interpreter **DSLInterpreter** **Description**: A domain-specific interpreter (DSL) **Python specific notes:** The object exposes a readable attribute 'DSLInterpreter'. This is the getter.
None	**Signature**: *[static,const]* Macro::Interpreter **None** **Description**: No specific interpreter **Python specific notes:** This member is available as 'None_' in Python. The object exposes a readable attribute 'None_'. This is the getter.
Python	**Signature**: *[static,const]* Macro::Interpreter **Python** **Description**: The interpreter is Python **Python specific notes:** The object exposes a readable attribute 'Python'. This is the getter.
Ruby	**Signature**: *[static,const]* Macro::Interpreter **Ruby** **Description**: The interpreter is Ruby **Python specific notes:** The object exposes a readable attribute 'Ruby'. This is the getter.
Text	**Signature**: *[static,const]* Macro::Interpreter **Text** **Description**: Plain text

Python specific notes:
The object exposes a readable attribute 'Text'. This is the getter.

hash

Signature: *[const]* int **hash**

Description: Gets the hash value from the enum

Python specific notes:
This method is also available as 'hash(object)'.

inspect

Signature: *[const]* string **inspect**

Description: Converts an enum to a visual string

Python specific notes:
This method is also available as 'repr(object)'.

new

(1) Signature: *[static]* new Macro::Interpreter ptr **new** (int i)

Description: Creates an enum from an integer value

Python specific notes:
This method is the default initializer of the object.

(2) Signature: *[static]* new Macro::Interpreter ptr **new** (string s)

Description: Creates an enum from a string value

Python specific notes:
This method is the default initializer of the object.

to_i

Signature: *[const]* int **to_i**

Description: Gets the integer value from the enum

Python specific notes:
This method is also available as 'int(object)'.

to_s

Signature: *[const]* string **to_s**

Description: Gets the symbolic string from an enum

Python specific notes:
This method is also available as 'str(object)'.

4.227. API reference - Class Annotation

Notation used in Ruby API documentation

Module: lay

Description: A layout annotation (i.e. ruler)

Class hierarchy: Annotation

Annotation objects provide a way to attach measurements or descriptive information to a layout view. Annotation objects can appear as rulers for example. Annotation objects can be configured in different ways using the styles provided. By configuring an annotation object properly, it can appear as a rectangle or a plain line for example. See Ruler properties for more details about the appearance options.

Annotations are inserted into a layout view using LayoutView#insert_annotation. Here is some sample code in Ruby:

```
app = RBA::Application.instance
mw = app.main_window
view = mw.current_view

ant = RBA::Annotation::new
ant.p1 = RBA::DPoint::new(0, 0)
ant.p2 = RBA::DPoint::new(100, 0)
ant.style = RBA::Annotation::StyleRuler
view.insert_annotation(ant)
```

Annotations can be retrieved from a view with LayoutView#each_annotation and all annotations can be cleared with LayoutView#clear_annotations.

Starting with version 0.25, annotations are 'live' objects once they are inserted into the view. Changing properties of annotations will automatically update the view (however, that is not true the other way round).

Here is some sample code of changing the style of all rulers to two-sided arrows:

```
view = RBA::LayoutView::current

begin

  view.transaction("Restyle annotations")

  view.each_annotation do |a|
    a.style = RBA::Annotation::StyleArrowBoth
  end

ensure
  view.commit
end
```

Public constructors

new Annotation ptr	**new**	Creates a new object of this class

Public methods

[const]	bool	**!=**	(const Annotation other)	Inequality operator

[const]	bool	==	(const Annotation other)	Equality operator
	void	_assign	(const Annotation other)	Assigns another object to self
	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	new Annotation ptr	_dup		Creates a copy of self
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
[const]	int	angle_constraint		Returns the angle constraint attribute
	void	angle_constraint=	(int flag)	Sets the angle constraint attribute
	void	assign	(const Annotation other)	Assigns another object to self
[const]	DBox	box		Gets the bounding box of the object (not including text)
[const]	string	category		Gets the category string
	void	category=	(string cat)	Sets the category string of the annotation
	void	delete		Deletes this annotation from the view
	void	detach		Detaches the annotation object from the view
[const]	new Annotation ptr	dup		Creates a copy of self
[const]	string	fmt		Returns the format used for the label
	void	fmt=	(string format)	Sets the format used for the label
[const]	string	fmt_x		Returns the format used for the x-axis label
	void	fmt_x=	(string format)	Sets the format used for the x-axis label
[const]	string	fmt_y		Returns the format used for the y-axis label

	void	**fmt_y=**	(string format)	Sets the format used for the y-axis label
[const]	int	**id**		Returns the annotation's ID
[const]	bool	**is_valid?**		Returns a value indicating whether the object is a valid reference.
[const]	int	**main_position**		Gets the position of the main label
	void	**main_position=**	(int pos)	Sets the position of the main label
[const]	int	**main_xalign**		Gets the horizontal alignment type of the main label
	void	**main_xalign=**	(int align)	Sets the horizontal alignment type of the main label
[const]	int	**main_yalign**		Gets the vertical alignment type of the main label
	void	**main_yalign=**	(int align)	Sets the vertical alignment type of the main label
[const]	int	**outline**		Returns the outline style of the annotation object
	void	**outline=**	(int outline)	Sets the outline style used for drawing the annotation object
[const]	DPoint	**p1**		Gets the first point of the ruler or marker
	void	**p1=**	(const DPoint point)	Sets the first point of the ruler or marker
[const]	DPoint	**p2**		Gets the second point of the ruler or marker
	void	**p2=**	(const DPoint point)	Sets the second point of the ruler or marker
[const]	DPoint[]	**points**		Gets the points of the ruler
	void	**points=**	(DPoint[] points)	Sets the points for a (potentially) multi-segmented ruler
[const]	DPoint	**seg_p1**	(unsigned long segment_index)	Gets the first point of the given segment.
[const]	DPoint	**seg_p2**	(unsigned long segment_index)	Gets the second point of the given segment.
[const]	unsigned long	**segments**		Gets the number of segments.
	void	**snap=**	(bool flag)	Sets the 'snap to objects' attribute
[const]	bool	**snap?**		Returns the 'snap to objects' attribute
[const]	int	**style**		Returns the style of the annotation object
	void	**style=**	(int style)	Sets the style used for drawing the annotation object

[const]	string	**text**	(unsigned long index = 0)	Returns the formatted text for the main label
[const]	string	**text_x**	(unsigned long index = 0)	Returns the formatted text for the x-axis label
[const]	string	**text_y**	(unsigned long index = 0)	Returns the formatted text for the y-axis label
[const]	string	**to_s**		Returns the string representation of the ruler
[const]	Annotation	**transformed**	(const DTrans t)	Transforms the ruler or marker with the given simple transformation
[const]	Annotation	**transformed**	(const DCplxTrans t)	Transforms the ruler or marker with the given complex transformation
[const]	Annotation	**transformed**	(const ICplxTrans t)	Transforms the ruler or marker with the given complex transformation
[const]	int	**xlabel_xalign**		Gets the horizontal alignment type of the x axis label
	void	**xlabel_xalign=**	(int align)	Sets the horizontal alignment type of the x axis label
[const]	int	**xlabel_yalign**		Gets the vertical alignment type of the x axis label
	void	**xlabel_yalign=**	(int align)	Sets the vertical alignment type of the x axis label
[const]	int	**ylabel_xalign**		Gets the horizontal alignment type of the y axis label
	void	**ylabel_xalign=**	(int align)	Sets the horizontal alignment type of the y axis label
[const]	int	**ylabel_yalign**		Gets the vertical alignment type of the y axis label
	void	**ylabel_yalign=**	(int align)	Sets the vertical alignment type of the y axis label

Public static methods and constants

int	**AlignAuto**	This code indicates automatic alignment.
int	**AlignBottom**	This code indicates bottom alignment.
int	**AlignCenter**	This code indicates automatic alignment.
int	**AlignDown**	This code indicates left or bottom alignment, depending on the context.
int	**AlignLeft**	This code indicates left alignment.

int	**AlignRight**	This code indicates right alignment.
int	**AlignTop**	This code indicates top alignment.
int	**AlignUp**	This code indicates right or top alignment, depending on the context.
int	**AngleAny**	Gets the any angle code for use with the angle_constraint method
int	**AngleDiagonal**	Gets the diagonal angle code for use with the angle_constraint method
int	**AngleGlobal**	Gets the global angle code for use with the angle_constraint method.
int	**AngleHorizontal**	Gets the horizontal angle code for use with the angle_constraint method
int	**AngleOrtho**	Gets the ortho angle code for use with the angle_constraint method
int	**AngleVertical**	Gets the vertical angle code for use with the angle_constraint method
int	**OutlineAngle**	Gets the angle measurement ruler outline code for use with the outline method
int	**OutlineBox**	Gets the box outline code for use with the outline method
int	**OutlineDiag**	Gets the diagonal output code for use with the outline method
int	**OutlineDiagXY**	outline_xy code used by the outline method
int	**OutlineDiagYX**	Gets the yx plus diagonal outline code for use with the outline method
int	**OutlineEllipse**	Gets the ellipse outline code for use with the outline method
int	**OutlineRadius**	Gets the radius measurement ruler outline code for use with the outline method
int	**OutlineXY**	Gets the xy outline code for use with the outline method
int	**OutlineYX**	Gets the yx outline code for use with the outline method
int	**PositionAuto**	This code indicates automatic positioning.
int	**PositionCenter**	This code indicates positioning of the main label at the mid point between p1 and p2.
int	**PositionP1**	This code indicates positioning of the main label at p1.
int	**PositionP2**	This code indicates positioning of the main label at p2.
int	**RulerModeAutoMetric**	Specifies auto-metric ruler mode for the register_template method

	int	**RulerModeAutoMetricEdge**		Specifies edge-sensitive auto-metric ruler mode for the register_template method
	int	**RulerModeNormal**		Specifies normal ruler mode for the register_template method
	int	**RulerModeSingleClick**		Specifies single-click ruler mode for the register_template method
	int	**RulerMultiSegment**		Specifies multi-segment mode
	int	**RulerThreeClicks**		Specifies three-click ruler mode for the register_template method
	int	**StyleArrowBoth**		Gets the both arrow ends style code for use the style method
	int	**StyleArrowEnd**		Gets the end arrow style code for use the style method
	int	**StyleArrowStart**		Gets the start arrow style code for use the style method
	int	**StyleCrossBoth**		Gets the line style code for use with the style method
	int	**StyleCrossEnd**		Gets the line style code for use with the style method
	int	**StyleCrossStart**		Gets the line style code for use with the style method
	int	**StyleLine**		Gets the line style code for use with the style method
	int	**StyleRuler**		Gets the ruler style code for use the style method
	new Annotation ptr	**from_s**	(string s)	Creates a ruler from a string representation
	void	**register_template**	(const Annotatic annotatic string title, int mode = RulerMo	Registers the given annotation as a template globally
	void	**unregister_templates**	(string category)	Unregisters the template or templates with the given category string globally

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**		Use of this method is deprecated. Use _create instead
	void	**destroy**		Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead

[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	Annotation	**transformed_cplx**	(const DCplxTrans t)	Use of this method is deprecated. Use transformed instead
[const]	Annotation	**transformed_cplx**	(const ICplxTrans t)	Use of this method is deprecated. Use transformed instead

Detailed description

!=

Signature: *[const]* bool **!=** (const Annotation other)

Description: Inequality operator

==

Signature: *[const]* bool **==** (const Annotation other)

Description: Equality operator

AlignAuto

Signature: *[static]* int **AlignAuto**

Description: This code indicates automatic alignment.

This code makes the annotation align the label the way it thinks is best.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'AlignAuto'. This is the getter.

AlignBottom

Signature: *[static]* int **AlignBottom**

Description: This code indicates bottom alignment.

If used in a vertical context, this alignment code makes the label aligned at the bottom side - i.e. it will appear top of the reference point.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'AlignBottom'. This is the getter.

AlignCenter

Signature: *[static]* int **AlignCenter**

Description: This code indicates automatic alignment.

This code makes the annotation align the label centered. When used in a horizontal context, centering is in horizontal direction. If used in a vertical context, centering is in vertical direction.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'AlignCenter'. This is the getter.

AlignDown

Signature: *[static]* int **AlignDown**

Description: This code indicates left or bottom alignment, depending on the context.

This code is equivalent to AlignLeft and AlignBottom.

This constant has been introduced in version 0.25.

Python specific notes:

The object exposes a readable attribute 'AlignDown'. This is the getter.

AlignLeft

Signature: *[static]* int **AlignLeft**

Description: This code indicates left alignment.

If used in a horizontal context, this alignment code makes the label aligned at the left side - i.e. it will appear right of the reference point.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'AlignLeft'. This is the getter.

AlignRight

Signature: *[static]* int **AlignRight**

Description: This code indicates right alignment.

If used in a horizontal context, this alignment code makes the label aligned at the right side - i.e. it will appear left of the reference point.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'AlignRight'. This is the getter.

AlignTop

Signature: *[static]* int **AlignTop**

Description: This code indicates top alignment.

If used in a vertical context, this alignment code makes the label aligned at the top side - i.e. it will appear bottom of the reference point.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'AlignTop'. This is the getter.

AlignUp

Signature: *[static]* int **AlignUp**

Description: This code indicates right or top alignment, depending on the context.

This code is equivalent to AlignRight and AlignTop.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'AlignUp'. This is the getter.

AngleAny

Signature: *[static]* int **AngleAny**

Description: Gets the any angle code for use with the angle_constraint method

If this value is specified for the angle constraint, all angles will be allowed.

Python specific notes:
The object exposes a readable attribute 'AngleAny'. This is the getter.

AngleDiagonal

Signature: *[static]* int **AngleDiagonal**

Description: Gets the diagonal angle code for use with the angle_constraint method

If this value is specified for the angle constraint, only multiples of 45 degree are allowed.

Python specific notes:
The object exposes a readable attribute 'AngleDiagonal'. This is the getter.

AngleGlobal

Signature: *[static]* int **AngleGlobal**

Description: Gets the global angle code for use with the angle_constraint method.

This code will tell the ruler or marker to use the angle constraint defined globally.

Python specific notes:
The object exposes a readable attribute 'AngleGlobal'. This is the getter.

AngleHorizontal

Signature: *[static]* int **AngleHorizontal**

Description: Gets the horizontal angle code for use with the angle_constraint method

If this value is specified for the angle constraint, only horizontal rulers are allowed.

Python specific notes:
The object exposes a readable attribute 'AngleHorizontal'. This is the getter.

AngleOrtho

Signature: *[static]* int **AngleOrtho**

Description: Gets the ortho angle code for use with the angle_constraint method

If this value is specified for the angle constraint, only multiples of 90 degree are allowed.

Python specific notes:
The object exposes a readable attribute 'AngleOrtho'. This is the getter.

AngleVertical

Signature: *[static]* int **AngleVertical**

Description: Gets the vertical angle code for use with the angle_constraint method

If this value is specified for the angle constraint, only vertical rulers are allowed.

Python specific notes:
The object exposes a readable attribute 'AngleVertical'. This is the getter.

OutlineAngle

Signature: *[static]* int **OutlineAngle**

Description: Gets the angle measurement ruler outline code for use with the outline method

When this outline style is specified, the ruler is drawn to indicate the angle between the first and last segment.

This constant has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'OutlineAngle'. This is the getter.

OutlineBox

Signature: *[static]* int **OutlineBox**

Description: Gets the box outline code for use with the outline method

When this outline style is specified, a box is drawn with the corners specified by the start and end point. All box edges are drawn in the style specified with the style attribute.

Python specific notes:
The object exposes a readable attribute 'OutlineBox'. This is the getter.

OutlineDiag

Signature: *[static]* int **OutlineDiag**

Description: Gets the diagonal output code for use with the outline method

When this outline style is specified, a line connecting start and end points in the given style (ruler, arrow or plain line) is drawn.

Python specific notes:
The object exposes a readable attribute 'OutlineDiag'. This is the getter.

OutlineDiagXY

Signature: *[static]* int **OutlineDiagXY**

Description: outline_xy code used by the outline method

When this outline style is specified, three lines are drawn: one horizontal from left to right and attached to the end of that a line from the bottom to the top. Another line is drawn connecting the start and end points directly. The lines are drawn in the specified style (see style method).

Python specific notes:
The object exposes a readable attribute 'OutlineDiagXY'. This is the getter.

OutlineDiagYX

Signature: *[static]* int **OutlineDiagYX**

Description: Gets the yx plus diagonal outline code for use with the outline method

When this outline style is specified, three lines are drawn: one vertical from bottom to top and attached to the end of that a line from the left to the right. Another line is drawn connecting the start and end points directly. The lines are drawn in the specified style (see style method).

Python specific notes:
The object exposes a readable attribute 'OutlineDiagYX'. This is the getter.

OutlineEllipse

Signature: *[static]* int **OutlineEllipse**

Description: Gets the ellipse outline code for use with the outline method

When this outline style is specified, an ellipse is drawn with the extensions specified by the start and end point. The contour drawn as a line.

This constant has been introduced in version 0.26.

Python specific notes:
The object exposes a readable attribute 'OutlineEllipse'. This is the getter.

OutlineRadius

Signature: *[static]* int **OutlineRadius**

Description: Gets the radius measurement ruler outline code for use with the outline method

When this outline style is specified, the ruler is drawn to indicate a radius defined by at least three points of the ruler.

This constant has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'OutlineRadius'. This is the getter.

OutlineXY

Signature: *[static]* int **OutlineXY**

Description: Gets the xy outline code for use with the outline method

When this outline style is specified, two lines are drawn: one horizontal from left to right and attached to the end of that a line from the bottom to the top. The lines are drawn in the specified style (see style method).

Python specific notes:
The object exposes a readable attribute 'OutlineXY'. This is the getter.

OutlineYX

Signature: *[static]* int **OutlineYX**

Description: Gets the yx outline code for use with the outline method

When this outline style is specified, two lines are drawn: one vertical from bottom to top and attached to the end of that a line from the left to the right. The lines are drawn in the specified style (see style method).

Python specific notes:
The object exposes a readable attribute 'OutlineYX'. This is the getter.

PositionAuto

Signature: *[static]* int **PositionAuto**

Description: This code indicates automatic positioning.

The main label will be put either to p1 or p2, whichever the annotation considers best.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'PositionAuto'. This is the getter.

PositionCenter

Signature: *[static]* int **PositionCenter**

Description: This code indicates positioning of the main label at the mid point between p1 and p2.

The main label will be put to the center point.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'PositionCenter'. This is the getter.

PositionP1

Signature: *[static]* int **PositionP1**

Description: This code indicates positioning of the main label at p1.

The main label will be put to p1.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'PositionP1'. This is the getter.

PositionP2

Signature: *[static]* int **PositionP2**

Description: This code indicates positioning of the main label at p2.

The main label will be put to p2.

This constant has been introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'PositionP2'. This is the getter.

RulerModeAutoMetric

Signature: *[static]* int **RulerModeAutoMetric**

Description: Specifies auto-metric ruler mode for the register_template method

In auto-metric mode, a ruler can be placed with a single click and p1/p2 will be determined from the neighborhood.

This constant has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'RulerModeAutoMetric'. This is the getter.

RulerModeAutoMetricEdg

Signature: *[static]* int **RulerModeAutoMetricEdge**

Description: Specifies edge-sensitive auto-metric ruler mode for the register_template method

In auto-metric mode, a ruler can be placed with a single click and p1/p2 will be determined from the edge it is placed on.

This constant has been introduced in version 0.29

Python specific notes:
The object exposes a readable attribute 'RulerModeAutoMetricEdge'. This is the getter.

RulerModeNormal

Signature: *[static]* int **RulerModeNormal**

Description: Specifies normal ruler mode for the register_template method

This constant has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'RulerModeNormal'. This is the getter.

RulerModeSingleClick

Signature: *[static]* int **RulerModeSingleClick**

Description: Specifies single-click ruler mode for the register_template method

In single click-mode, a ruler can be placed with a single click and p1 will be == p2.

This constant has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'RulerModeSingleClick'. This is the getter.

RulerMultiSegment

Signature: *[static]* int **RulerMultiSegment**

Description: Specifies multi-segment mode

In multi-segment mode, multiple segments can be created. The ruler is finished with a double click.

This constant has been introduced in version 0.28

Python specific notes:
The object exposes a readable attribute 'RulerMultiSegment'. This is the getter.

RulerThreeClicks

Signature: *[static]* int **RulerThreeClicks**

Description: Specifies three-click ruler mode for the register_template method

In this ruler mode, two segments are created for angle and circle radius measurements. Three mouse clicks are required.

This constant has been introduced in version 0.28

Python specific notes:
The object exposes a readable attribute 'RulerThreeClicks'. This is the getter.

StyleArrowBoth

Signature: *[static]* int **StyleArrowBoth**

Description: Gets the both arrow ends style code for use the style method

When this style is specified, a two-headed arrow is drawn.

Python specific notes:
The object exposes a readable attribute 'StyleArrowBoth'. This is the getter.

StyleArrowEnd

Signature: *[static]* int **StyleArrowEnd**

Description: Gets the end arrow style code for use the style method

When this style is specified, an arrow is drawn pointing from the start to the end point.

Python specific notes:
The object exposes a readable attribute 'StyleArrowEnd'. This is the getter.

StyleArrowStart

Signature: *[static]* int **StyleArrowStart**

Description: Gets the start arrow style code for use the style method

When this style is specified, an arrow is drawn pointing from the end to the start point.

Python specific notes:
The object exposes a readable attribute 'StyleArrowStart'. This is the getter.

StyleCrossBoth

Signature: *[static]* int **StyleCrossBoth**

Description: Gets the line style code for use with the style method

When this style is specified, a cross is drawn at both points.

This constant has been added in version 0.26.

Python specific notes:
The object exposes a readable attribute 'StyleCrossBoth'. This is the getter.

StyleCrossEnd

Signature: *[static]* int **StyleCrossEnd**

Description: Gets the line style code for use with the style method

When this style is specified, a cross is drawn at the end point.

This constant has been added in version 0.26.

Python specific notes:
The object exposes a readable attribute 'StyleCrossEnd'. This is the getter.

StyleCrossStart

Signature: *[static]* int **StyleCrossStart**

Description: Gets the line style code for use with the style method

When this style is specified, a cross is drawn at the start point.

This constant has been added in version 0.26.

Python specific notes:
The object exposes a readable attribute 'StyleCrossStart'. This is the getter.

StyleLine

Signature: *[static]* int **StyleLine**

Description: Gets the line style code for use with the style method

When this style is specified, a plain line is drawn.

Python specific notes:
The object exposes a readable attribute 'StyleLine'. This is the getter.

StyleRuler

Signature: *[static]* int **StyleRuler**

Description: Gets the ruler style code for use the style method

When this style is specified, the annotation will show a ruler with some ticks at distances indicating a decade of units and a suitable subdivision into minor ticks at intervals of 1, 2 or 5 units.

Python specific notes:
The object exposes a readable attribute 'StyleRuler'. This is the getter.

_assign

Signature: void **_assign** (const Annotation other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new Annotation ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

angle_constraint

Signature: *[const]* int **angle_constraint**

Description: Returns the angle constraint attribute

See angle_constraint= for a more detailed description.

Python specific notes:
The object exposes a readable attribute 'angle_constraint'. This is the getter.

angle_constraint=

Signature: void **angle_constraint=** (int flag)

Description: Sets the angle constraint attribute

This attribute controls if an angle constraint is applied when moving one of the ruler's points. The Angle... values can be used for this purpose.

Python specific notes:
The object exposes a writable attribute 'angle_constraint'. This is the setter.

assign

Signature: void **assign** (const Annotation other)

Description: Assigns another object to self

box

Signature: *[const]* DBox **box**

Description: Gets the bounding box of the object (not including text)

Returns: The bounding box

category

Signature: *[const]* string **category**

Description: Gets the category string

See category= for details.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'category'. This is the getter.

category=

Signature: void **category=** (string cat)

Description: Sets the category string of the annotation

The category string is an arbitrary string that can be used by various consumers or generators to mark 'their' annotation.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a writable attribute 'category'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

delete

Signature: void **delete**

Description: Deletes this annotation from the view

If the annotation is an "active" one, this method will remove it from the view. This object will become detached and can still be manipulated, but without having an effect on the view. This method has been introduced in version 0.25.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

detach

Signature: void **detach**

Description: Detaches the annotation object from the view

If the annotation object was inserted into the view, property changes will be reflected in the view. To disable this feature, 'detach' can be called after which the annotation object becomes inactive and changes will no longer be reflected in the view.

This method has been introduced in version 0.25.

dup

Signature: *[const]* new Annotation ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

fmt

Signature: *[const]* string **fmt**

Description: Returns the format used for the label

Returns: The format string

Format strings can contain placeholders for values and formulas for computing derived values. See Ruler properties for more details.

Python specific notes:
The object exposes a readable attribute 'fmt'. This is the getter.

fmt=

Signature: void **fmt=** (string format)

Description: Sets the format used for the label

format: The format string

Format strings can contain placeholders for values and formulas for computing derived values. See Ruler properties for more details.

Python specific notes:
The object exposes a writable attribute 'fmt'. This is the setter.

fmt_x

Signature: *[const]* string **fmt_x**

Description: Returns the format used for the x-axis label

Returns: The format string

Format strings can contain placeholders for values and formulas for computing derived values. See Ruler properties for more details.

Python specific notes:
The object exposes a readable attribute 'fmt_x'. This is the getter.

fmt_x=

Signature: void **fmt_x=** (string format)

Description: Sets the format used for the x-axis label

format: The format string

X-axis labels are only used for styles that have a horizontal component. Format strings can contain placeholders for values and formulas for computing derived values. See Ruler properties for more details.

Python specific notes:

The object exposes a writable attribute 'fmt_x'. This is the setter.

fmt_y

Signature: *[const]* string **fmt_y**

Description: Returns the format used for the y-axis label

Returns: The format string

Format strings can contain placeholders for values and formulas for computing derived values. See Ruler properties for more details.

Python specific notes:
The object exposes a readable attribute 'fmt_y'. This is the getter.

fmt_y=

Signature: void **fmt_y=** (string format)

Description: Sets the format used for the y-axis label

format: The format string

Y-axis labels are only used for styles that have a vertical component. Format strings can contain placeholders for values and formulas for computing derived values. See Ruler properties for more details.

Python specific notes:
The object exposes a writable attribute 'fmt_y'. This is the setter.

from_s

Signature: *[static]* new Annotation ptr **from_s** (string s)

Description: Creates a ruler from a string representation

This function creates a ruler from the string returned by to_s.

This method was introduced in version 0.28.

id

Signature: *[const]* int **id**

Description: Returns the annotation's ID

The annotation ID is an integer that uniquely identifies an annotation inside a view. The ID is used for replacing an annotation (see LayoutView#replace_annotation).

This method was introduced in version 0.24.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_valid?

Signature: *[const]* bool **is_valid?**

Description: Returns a value indicating whether the object is a valid reference.

If this value is true, the object represents an annotation on the screen. Otherwise, the object is a 'detached' annotation which does not have a representation on the screen.

This method was introduced in version 0.25.

main_position

Signature: *[const]* int **main_position**

Description: Gets the position of the main label

See main_position= for details.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'main_position'. This is the getter.

main_position=

Signature: void **main_position=** (int pos)

Description: Sets the position of the main label

This method accepts one of the Position... constants.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a writable attribute 'main_position'. This is the setter.

main_xalign

Signature: *[const]* int **main_xalign**

Description: Gets the horizontal alignment type of the main label

See main_xalign= for details.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'main_xalign'. This is the getter.

main_xalign=

Signature: void **main_xalign=** (int align)

Description: Sets the horizontal alignment type of the main label

This method accepts one of the Align... constants.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a writable attribute 'main_xalign'. This is the setter.

main_yalign

Signature: *[const]* int **main_yalign**

Description: Gets the vertical alignment type of the main label

See main_yalign= for details.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'main_yalign'. This is the getter.

main_yalign=

Signature: void **main_yalign=** (int align)

Description: Sets the vertical alignment type of the main label

This method accepts one of the Align... constants.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a writable attribute 'main_yalign'. This is the setter.

new

Signature: *[static]* new Annotation ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

outline

Signature: *[const]* int **outline**

Description: Returns the outline style of the annotation object

Python specific notes:

The object exposes a readable attribute 'outline'. This is the getter.

outline=

Signature: void **outline=** (int outline)

Description: Sets the outline style used for drawing the annotation object

The Outline... values can be used for defining the annotation object's outline. The outline style determines what components are drawn.

Python specific notes:
The object exposes a writable attribute 'outline'. This is the setter.

p1

Signature: *[const]* DPoint **p1**

Description: Gets the first point of the ruler or marker

Returns: The first point

The points of the ruler or marker are always given in micron units in floating-point coordinates.

This method is provided for backward compatibility. Starting with version 0.28, rulers can be multi-segmented. Use points or seg_p1 to retrieve the points of the ruler segments.

Python specific notes:
The object exposes a readable attribute 'p1'. This is the getter.

p1=

Signature: void **p1=** (const DPoint point)

Description: Sets the first point of the ruler or marker

The points of the ruler or marker are always given in micron units in floating-point coordinates.

This method is provided for backward compatibility. Starting with version 0.28, rulers can be multi-segmented. Use points= to specify the ruler segments.

Python specific notes:
The object exposes a writable attribute 'p1'. This is the setter.

p2

Signature: *[const]* DPoint **p2**

Description: Gets the second point of the ruler or marker

Returns: The second point

The points of the ruler or marker are always given in micron units in floating-point coordinates.

This method is provided for backward compatibility. Starting with version 0.28, rulers can be multi-segmented. Use points or seg_p1 to retrieve the points of the ruler segments.

Python specific notes:
The object exposes a readable attribute 'p2'. This is the getter.

p2=

Signature: void **p2=** (const DPoint point)

Description: Sets the second point of the ruler or marker

The points of the ruler or marker are always given in micron units in floating-point coordinates.

This method is provided for backward compatibility. Starting with version 0.28, rulers can be multi-segmented. Use points= to specify the ruler segments.

Python specific notes:
The object exposes a writable attribute 'p2'. This is the setter.

points

Signature: *[const]* DPoint[] **points**

Description: Gets the points of the ruler

A single-segmented ruler has two points. Rulers with more points have more segments correspondingly. Note that the point list may have one point only (single-point ruler) or may even be empty.

Use points= to set the segment points. Use segments to get the number of segments and seg_p1 and seg_p2 to get the first and second point of one segment.

Multi-segmented rulers have been introduced in version 0.28

Python specific notes:
The object exposes a readable attribute 'points'. This is the getter.

points=

Signature: void **points=** (DPoint[] points)

Description: Sets the points for a (potentially) multi-segmented ruler

See points for a description of multi-segmented rulers. The list of points passed to this method is cleaned from duplicates before being stored inside the ruler.

This method has been introduced in version 0.28.

Python specific notes:
The object exposes a writable attribute 'points'. This is the setter.

register_template

Signature: *[static]* void **register_template** (const Annotation annotation, string title, int mode = RulerModeNormal)

Description: Registers the given annotation as a template globally

title:	The title to use for the ruler template
mode:	The mode the ruler will be created in (see Ruler... constants)

@annotation The annotation to use for the template (positions are ignored)

In order to register a system template, the category string of the annotation has to be a unique and non-empty string. The annotation is added to the list of annotation templates and becomes available as a new template in the ruler drop-down menu.

The new annotation template is registered on all views.

NOTE: this setting is persisted and the the application configuration is updated.

This method has been added in version 0.25.

seg_p1

Signature: *[const]* DPoint **seg_p1** (unsigned long segment_index)

Description: Gets the first point of the given segment.

The segment is indicated by the segment index which is a number between 0 and segments-1.

This method has been introduced in version 0.28.

seg_p2

Signature: *[const]* DPoint **seg_p2** (unsigned long segment_index)

Description: Gets the second point of the given segment.

The segment is indicated by the segment index which is a number between 0 and segments-1. The second point of a segment is also the first point of the following segment if there is one.

This method has been introduced in version 0.28.

segments

Signature: *[const]* unsigned long **segments**

Description: Gets the number of segments.

This method returns the number of segments the ruler is made up. Even though the ruler can be one or even zero points, the number of segments is at least 1.

This method has been introduced in version 0.28.

snap=

Signature: void **snap=** (bool flag)

Description: Sets the 'snap to objects' attribute

If this attribute is set to true, the ruler or marker snaps to other objects when moved.

Python specific notes:
The object exposes a writable attribute 'snap'. This is the setter.

snap?

Signature: *[const]* bool **snap?**

Description: Returns the 'snap to objects' attribute

Python specific notes:
The object exposes a readable attribute 'snap'. This is the getter.

style

Signature: *[const]* int **style**

Description: Returns the style of the annotation object

Python specific notes:
The object exposes a readable attribute 'style'. This is the getter.

style=

Signature: void **style=** (int style)

Description: Sets the style used for drawing the annotation object

The Style... values can be used for defining the annotation object's style. The style determines if ticks or arrows are drawn.

Python specific notes:
The object exposes a writable attribute 'style'. This is the setter.

text

Signature: *[const]* string **text** (unsigned long index = 0)

Description: Returns the formatted text for the main label

The index parameter indicates which segment to use (0 is the first one). It has been added in version 0.28.

text_x

Signature: *[const]* string **text_x** (unsigned long index = 0)

Description: Returns the formatted text for the x-axis label

The index parameter indicates which segment to use (0 is the first one). It has been added in version 0.28.

text_y

Signature: *[const]* string **text_y** (unsigned long index = 0)

Description: Returns the formatted text for the y-axis label

The index parameter indicates which segment to use (0 is the first one). It has been added in version 0.28.

to_s

Signature: *[const]* string **to_s**

Description: Returns the string representation of the ruler

This method was introduced in version 0.19.

Python specific notes:
This method is also available as 'str(object)'.

transformed

(1) Signature: *[const]* Annotation **transformed** (const DTrans t)

Description: Transforms the ruler or marker with the given simple transformation

t: The transformation to apply

Returns: The transformed object

(2) Signature: *[const]* Annotation **transformed** (const DCplxTrans t)

Description: Transforms the ruler or marker with the given complex transformation

t: The magnifying transformation to apply

Returns: The transformed object

Starting with version 0.25, all overloads all available as 'transform'.

(3) Signature: *[const]* Annotation **transformed** (const ICplxTrans t)

Description: Transforms the ruler or marker with the given complex transformation

t: The magnifying transformation to apply

Returns: The transformed object (in this case an integer coordinate object)

This method has been introduced in version 0.18.

Starting with version 0.25, all overloads all available as 'transform'.

transformed_cplx

(1) Signature: *[const]* Annotation **transformed_cplx** (const DCplxTrans t)

Description: Transforms the ruler or marker with the given complex transformation

t: The magnifying transformation to apply

Returns: The transformed object

Use of this method is deprecated. Use transformed instead

Starting with version 0.25, all overloads all available as 'transform'.

(2) Signature: *[const]* Annotation **transformed_cplx** (const ICplxTrans t)

Description: Transforms the ruler or marker with the given complex transformation

t: The magnifying transformation to apply

Returns: The transformed object (in this case an integer coordinate object)

Use of this method is deprecated. Use transformed instead

This method has been introduced in version 0.18.

Starting with version 0.25, all overloads all available as 'transform'.

unregister_templates

Signature: *[static]* void **unregister_templates** (string category)

Description: Unregisters the template or templates with the given category string globally

This method will remove all templates with the given category string. If the category string is empty, all templates are removed.

NOTE: this setting is persisted and the the application configuration is updated.

This method has been added in version 0.28.

xlabel_xalign

Signature: *[const]* int **xlabel_xalign**

Description: Gets the horizontal alignment type of the x axis label

See xlabel_xalign= for details.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'xlabel_xalign'. This is the getter.

xlabel_xalign=

Signature: void **xlabel_xalign=** (int align)

Description: Sets the horizontal alignment type of the x axis label

This method accepts one of the Align... constants.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a writable attribute 'xlabel_xalign'. This is the setter.

xlabel_yalign

Signature: *[const]* int **xlabel_yalign**

Description: Gets the vertical alignment type of the x axis label

See xlabel_yalign= for details.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'xlabel_yalign'. This is the getter.

xlabel_yalign=

Signature: void **xlabel_yalign=** (int align)

Description: Sets the vertical alignment type of the x axis label

This method accepts one of the Align... constants.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a writable attribute 'xlabel_yalign'. This is the setter.

ylabel_xalign

Signature: *[const]* int **ylabel_xalign**

Description: Gets the horizontal alignment type of the y axis label

See ylabel_xalign= for details.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'ylabel_xalign'. This is the getter.

ylabel_xalign=

Signature: void **ylabel_xalign=** (int align)

Description: Sets the horizontal alignment type of the y axis label

This method accepts one of the Align... constants.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a writable attribute 'ylabel_xalign'. This is the setter.

ylabel_yalign

Signature: *[const]* int **ylabel_yalign**

Description: Gets the vertical alignment type of the y axis label

See ylabel_yalign= for details.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a readable attribute 'ylabel_yalign'. This is the getter.

ylabel_yalign=

Signature: void **ylabel_yalign=** (int align)

Description: Sets the vertical alignment type of the y axis label

This method accepts one of the Align... constants.

This method has been introduced in version 0.25

Python specific notes:
The object exposes a writable attribute 'ylabel_yalign'. This is the setter.

4.228. API reference - Class ImageDataMapping

Notation used in Ruby API documentation

Module: lay

Description: A structure describing the data mapping of an image object

Data mapping is the process of transforming the data into RGB pixel values. This implementation provides four adjustment steps: first, in the case of monochrome data, the data is converted to a RGB triplet using the color map. The default color map will copy the value to all channels rendering a gray scale. After having normalized the data to 0..1 cooresponding to the min_value and max_value settings of the image, a color channel-independent brightness and contrast adjustment is applied. Then, a per-channel multiplier (red_gain, green_gain, blue_gain) is applied. Finally, the gamma function is applied and the result converted into a 0..255 pixel value range and clipped.

Public constructors

new ImageDataMapping ptr	**new**	Create a new data mapping object with default settings

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add_colormap_entry**	(double value, unsigned int color)	Add a colormap entry for this data mapping object.
	void	**add_colormap_entry**	(double value, unsigned int lcolor, unsigned int rcolor)	Add a colormap entry for this data mapping object.
	void	**assign**	(const ImageDataMapping other)	Assigns another object to self
[const]	double	**blue_gain**		The blue channel gain
	void	**blue_gain=**	(double blue_gain)	Set the blue_gain
[const]	double	**brightness**		The brightness value
	void	**brightness=**	(double brightness)	Set the brightness

	void	clear_colormap		The the color map of this data mapping object.
[const]	unsigned int	colormap_color	(unsigned long n)	Returns the color for a given color map entry.
[const]	unsigned int	colormap_lcolor	(unsigned long n)	Returns the left-side color for a given color map entry.
[const]	unsigned int	colormap_rcolor	(unsigned long n)	Returns the right-side color for a given color map entry.
[const]	double	colormap_value	(unsigned long n)	Returns the value for a given color map entry.
[const]	double	contrast		The contrast value
	void	contrast=	(double contrast)	Set the contrast
[const]	new ImageDataMapping ptr	dup		Creates a copy of self
[const]	double	gamma		The gamma value
	void	gamma=	(double gamma)	Set the gamma
[const]	double	green_gain		The green channel gain
	void	green_gain=	(double green_gain)	Set the green_gain
[const]	unsigned long	num_colormap_entries		Returns the current number of color map entries.
[const]	double	red_gain		The red channel gain
	void	red_gain=	(double red_gain)	Set the red_gain

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_colormap_entry

(1) Signature: void **add_colormap_entry** (double value, unsigned int color)

Description: Add a colormap entry for this data mapping object.

value: The value at which the given color should be applied.

color: The color to apply (a 32 bit RGB value).

This settings establishes a color mapping for a given value in the monochrome channel. The color must be given as a 32 bit integer, where the lowest order byte describes the blue component (0 to 255), the second byte the green component and the third byte the red component, i.e. 0xff0000 is red and 0x0000ff is blue.

(2) Signature: void **add_colormap_entry** (double value, unsigned int lcolor, unsigned int rcolor)

Description: Add a colormap entry for this data mapping object.

value:	The value at which the given color should be applied.
lcolor:	The color to apply left of the value (a 32 bit RGB value).
rcolor:	The color to apply right of the value (a 32 bit RGB value).

This settings establishes a color mapping for a given value in the monochrome channel. The colors must be given as a 32 bit integer, where the lowest order byte describes the blue component (0 to 255), the second byte the green component and the third byte the red component, i.e. 0xff0000 is red and 0x0000ff is blue.

In contrast to the version with one color, this version allows specifying a color left and right of the value - i.e. a discontinuous step.

This variant has been introduced in version 0.27.

assign

Signature: void **assign** (const ImageDataMapping other)

Description: Assigns another object to self

blue_gain

Signature: *[const]* double **blue_gain**

Description: The blue channel gain

This value is the multiplier by which the blue channel is scaled after applying false color transformation and contrast/brightness/gamma.

1.0 is a neutral value. The gain should be >=0.0.

Python specific notes:
The object exposes a readable attribute 'blue_gain'. This is the getter.

blue_gain=

Signature: void **blue_gain=** (double blue_gain)

Description: Set the blue_gain

See blue_gain for a description of this property.

Python specific notes:
The object exposes a writable attribute 'blue_gain'. This is the setter.

brightness

Signature: *[const]* double **brightness**

Description: The brightness value

The brightness is a double value between roughly -1.0 and 1.0. Neutral (original) brightness is 0.0.

Python specific notes:
The object exposes a readable attribute 'brightness'. This is the getter.

brightness=

Signature: void **brightness=** (double brightness)

Description: Set the brightness

See brightness for a description of this property.

Python specific notes:
The object exposes a writable attribute 'brightness'. This is the setter.

clear_colormap

Signature: void **clear_colormap**

Description: The the color map of this data mapping object.

colormap_color

Signature: *[const]* unsigned int **colormap_color** (unsigned long n)

Description: Returns the color for a given color map entry.

n: The index of the entry (0..num_colormap_entries-1)

Returns: The color (see add_colormap_entry for a description).

NOTE: this version is deprecated and provided for backward compatibility. For discontinuous nodes this method delivers the left-sided color.

colormap_lcolor

Signature: *[const]* unsigned int **colormap_lcolor** (unsigned long n)

Description: Returns the left-side color for a given color map entry.

n: The index of the entry (0..num_colormap_entries-1)

Returns: The color (see add_colormap_entry for a description).

This method has been introduced in version 0.27.

colormap_rcolor

Signature: *[const]* unsigned int **colormap_rcolor** (unsigned long n)

Description: Returns the right-side color for a given color map entry.

n: The index of the entry (0..num_colormap_entries-1)

Returns: The color (see add_colormap_entry for a description).

This method has been introduced in version 0.27.

colormap_value

Signature: *[const]* double **colormap_value** (unsigned long n)

Description: Returns the value for a given color map entry.

n: The index of the entry (0..num_colormap_entries-1)

Returns: The value (see add_colormap_entry for a description).

contrast

Signature: *[const]* double **contrast**

Description: The contrast value

The contrast is a double value between roughly -1.0 and 1.0. Neutral (original) contrast is 0.0.

Python specific notes:
The object exposes a readable attribute 'contrast'. This is the getter.

contrast=

Signature: void **contrast=** (double contrast)

Description: Set the contrast

See contrast for a description of this property.

Python specific notes:
The object exposes a writable attribute 'contrast'. This is the setter.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new ImageDataMapping ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

gamma

Signature: *[const]* double **gamma**

Description: The gamma value

The gamma value allows one to adjust for non-linearities in the display chain and to enhance contrast. A value for linear intensity reproduction on the screen is roughly 0.5. The exact value depends on the monitor calibration. Values below 1.0 give a "softer" appearance while values above 1.0 give a "harder" appearance.

Python specific notes:
The object exposes a readable attribute 'gamma'. This is the getter.

gamma=

Signature: void **gamma=** (double gamma)

Description: Set the gamma

See gamma for a description of this property.

Python specific notes:
The object exposes a writable attribute 'gamma'. This is the setter.

green_gain

Signature: *[const]* double **green_gain**

Description: The green channel gain

This value is the multiplier by which the green channel is scaled after applying false color transformation and contrast/brightness/gamma.

1.0 is a neutral value. The gain should be >=0.0.

Python specific notes:
The object exposes a readable attribute 'green_gain'. This is the getter.

green_gain=

Signature: void **green_gain=** (double green_gain)

Description: Set the green_gain

See green_gain for a description of this property.

Python specific notes:

	The object exposes a writable attribute 'green_gain'. This is the setter.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
new	**Signature**: *[static]* new ImageDataMapping ptr **new** **Description**: Create a new data mapping object with default settings **Python specific notes:** This method is the default initializer of the object.
num_colormap_entries	**Signature**: *[const]* unsigned long **num_colormap_entries** **Description**: Returns the current number of color map entries. **Returns**: The number of entries.
red_gain	**Signature**: *[const]* double **red_gain** **Description**: The red channel gain This value is the multiplier by which the red channel is scaled after applying false color transformation and contrast/brightness/gamma. 1.0 is a neutral value. The gain should be >=0.0. **Python specific notes:** The object exposes a readable attribute 'red_gain'. This is the getter.
red_gain=	**Signature**: void **red_gain=** (double red_gain) **Description**: Set the red_gain See red_gain for a description of this property. **Python specific notes:** The object exposes a writable attribute 'red_gain'. This is the setter.

4.229. API reference - Class Image

Notation used in Ruby API documentation

Module: lay

Description: An image to be stored as a layout annotation

Class hierarchy: Image

Images can be put onto the layout canvas as annotations, along with rulers and markers. Images can be monochrome (represent scalar data) as well as color (represent color images). The display of images can be adjusted in various ways, i.e. color mapping (translation of scalar values to colors), geometrical transformations (including rotation by arbitrary angles) and similar. Images are always based on floating point data. The actual data range is not fixed and can be adjusted to the data set (i.e. 0..255 or -1..1). This gives a great flexibility when displaying data which is the result of some measurement or calculation for example. The basic parameters of an image are the width and height of the data set, the width and height of one pixel, the geometrical transformation to be applied, the data range (min_value to max_value) and the data mapping which is described by an own class, ImageDataMapping.

Starting with version 0.22, the basic transformation is a 3x3 matrix rather than the simple affine transformation. This matrix includes the pixel dimensions as well. One consequence of that is that the magnification part of the matrix and the pixel dimensions are no longer separated. That has certain consequences, i.e. setting an affine transformation with a magnification scales the pixel sizes as before but an affine transformation returned will no longer contain the pixel dimensions as magnification because it only supports isotropic scaling. For backward compatibility, the rotation center for the affine transformations while the default center and the center for matrix transformations is the image center.

As with version 0.25, images become 'live' objects. Changes to image properties will be reflected in the view automatically once the image object has been inserted into a view. Note that changes are not immediately reflected in the view, but are delayed until the view is refreshed. Hence, iterating the view's images will not render the same results than the image objects attached to the view. To ensure synchronization, call Image#update.

Public constructors

new Image ptr	**new**		Create a new image with the default attributes
new Image ptr	**new**	(string filename, const DCplxTrans trans = unity)	Constructor from a image file
new Image ptr	**new**	(const PixelBuffer pixels, const DCplxTrans trans = unity)	Constructor from a image pixel buffer
new Image ptr	**new**	(const QImage image, const DCplxTrans trans = unity)	Constructor from a image pixel buffer
new Image ptr	**new**	(unsigned long w, unsigned long h, double[] data)	Constructor for a monochrome image with the given pixel values
new Image ptr	**new**	(unsigned long w, unsigned long h, const DCplxTrans trans, double[] data)	Constructor for a monochrome image with the given pixel values
new Image ptr	**new**	(unsigned long w, unsigned long h, double[] red, double[] green, double[] blue)	Constructor for a color image with the given pixel values
new Image ptr	**new**	(unsigned long w, unsigned long h, const DCplxTrans trans,	Constructor for a color image with the given pixel values

double[] red,
double[] green,
double[] blue)

Public methods

	void	_assign	(const Image other)	Assigns another object to self
	void	_create		Ensures the C++ object is created
	void	_destroy		Explicitly destroys the object
[const]	bool	_destroyed?		Returns a value indicating whether the object was already destroyed
[const]	new Image ptr	_dup		Creates a copy of self
[const]	bool	_is_const_object?		Returns a value indicating whether the reference is a const reference
	void	_manage		Marks the object as managed by the script side.
	void	_unmanage		Marks the object as no longer owned by the script side.
	void	assign	(const Image other)	Assigns another object to self
[const]	DBox	box		Gets the bounding box of the image
	void	clear		Clears the image data (sets to 0 or black).
	double[]	data	(int channel = 0)	Gets the data array for a specific color channel
[const]	ImageDataMapping	data_mapping		Gets the data mapping
	void	data_mapping=	(const ImageDataMapping data_mapping)	Sets the data mapping object
	void	delete		Deletes this image from the view
	void	detach		Detaches the image object from the view
[const]	new Image ptr	dup		Creates a copy of self
[const]	string	filename		Gets the name of the file loaded of an empty string if not file is loaded
[const]	double	get_pixel	(unsigned long x, unsigned long y)	Gets one pixel (monochrome only)
[const]	double	get_pixel	(unsigned long x, unsigned long y,	Gets one pixel (monochrome and color)

			unsigned int component)	
[const]	unsigned long	**height**		Gets the height of the image in pixels
[const]	unsigned long	**id**		Gets the Id
[const]	bool	**is_color?**		Returns true, if the image is a color image
[const]	bool	**is_empty?**		Returns true, if the image does not contain any data (i.e. is default constructed)
[const]	bool	**is_valid?**		Returns a value indicating whether the object is a valid reference.
[const]	bool	**is_visible?**		Gets a flag indicating whether the image object is visible
[const]	bool	**mask**	(unsigned long x, unsigned long y)	Gets the mask for one pixel
	bool[]	**mask_data**		Gets the mask from a array of boolean values
	void	**mask_data=**	(bool[] mask_data)	Sets the mask from a array of boolean values
[const]	Matrix3d	**matrix**		Returns the pixel-to-micron transformation matrix
	void	**matrix=**	(const Matrix3d t)	Sets the transformation matrix
[const]	double	**max_value**		Sets the maximum value
	void	**max_value=**	(double v)	Gets the upper limit of the values in the data set
[const]	double	**min_value**		Gets the upper limit of the values in the data set
	void	**min_value=**	(double v)	Sets the minimum value
[const]	double	**pixel_height**		Gets the pixel height
	void	**pixel_height=**	(double h)	Sets the pixel height
[const]	double	**pixel_width**		Gets the pixel width
	void	**pixel_width=**	(double w)	Sets the pixel width
	void	**set_data**	(unsigned long w, unsigned long h, double[] d)	Writes the image data field (monochrome)
	void	**set_data**	(unsigned long w, unsigned long h, double[] r, double[] g, double[] b)	Writes the image data field (color)

	void	**set_mask**	(unsigned long x, unsigned long y, bool m)	Sets the mask for a pixel
	void	**set_pixel**	(unsigned long x, unsigned long y, double v)	Sets one pixel (monochrome)
	void	**set_pixel**	(unsigned long x, unsigned long y, double r, double g, double b)	Sets one pixel (color)
[const]	string	**to_s**		Converts the image to a string
[const]	DCplxTrans	**trans**		Returns the pixel-to-micron transformation
	void	**trans=**	(const DCplxTrans t)	Sets the transformation
[const]	Image	**transformed**	(const DTrans t)	Transforms the image with the given simple transformation
[const]	Image	**transformed**	(const Matrix3d t)	Transforms the image with the given matrix transformation
[const]	Image	**transformed**	(const DCplxTrans t)	Transforms the image with the given complex transformation
	void	**update**		Forces an update of the view
	void	**visible=**	(bool v)	Sets the visibility
[const]	unsigned long	**width**		Gets the width of the image in pixels
[const]	void	**write**	(string path)	Saves the image to KLayout's image format (.lyimg)
[const]	int	**z_position**		Gets the z position of the image
	void	**z_position=**	(int z)	Sets the z position of the image

Public static methods and constants

new Image ptr	**from_s**	(string s)	Creates an image from the string returned by to_s.
new Image ptr	**read**	(string path)	Loads the image from the given path.

Deprecated methods (protected, public, static, non-static and constructors)

void	**create**		Use of this method is deprecated. Use _create instead
void	**destroy**		Use of this method is deprecated. Use _destroy instead

[const]	bool	**destroyed?**		Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**		Use of this method is deprecated. Use _is_const_object? instead
[const]	Image	**transformed_cplx**	(const DCplxTrans t)	Use of this method is deprecated. Use transformed instead
[const]	Image	**transformed_matrix**	(const Matrix3d t)	Use of this method is deprecated. Use transformed instead

Detailed description

_assign

Signature: void **_assign** (const Image other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new Image ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const Image other)

Description: Assigns another object to self

box

Signature: *[const]* DBox **box**

Description: Gets the bounding box of the image

Returns: The bounding box

clear

Signature: void **clear**

Description: Clears the image data (sets to 0 or black).

This method has been introduced in version 0.27.

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

data

Signature: double[] **data** (int channel = 0)

Description: Gets the data array for a specific color channel

Returns an array of pixel values for the given channel. For a color image, channel 0 is green, channel 1 is red and channel 2 is blue. For a monochrome image, the channel is ignored.

For the format of the data see the constructor description.

This method has been introduced in version 0.27.

data_mapping

Signature: *[const]* ImageDataMapping **data_mapping**

Description: Gets the data mapping

Returns: The data mapping object

The data mapping describes the transformation of a pixel value (any double value) into pixel data which can be sent to the graphics cards for display. See ImageDataMapping for a more detailed description.

Python specific notes:
The object exposes a readable attribute 'data_mapping'. This is the getter.

data_mapping=

Signature: void **data_mapping=** (const ImageDataMapping data_mapping)

Description: Sets the data mapping object

The data mapping describes the transformation of a pixel value (any double value) into pixel data which can be sent to the graphics cards for display. See ImageDataMapping for a more detailed description.

Python specific notes:
The object exposes a writable attribute 'data_mapping'. This is the setter.

delete

Signature: void **delete**

Description: Deletes this image from the view

If the image is an "active" one, this method will remove it from the view. This object will become detached and can still be manipulated, but without having an effect on the view. This method has been introduced in version 0.25.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

detach

Signature: void **detach**

Description: Detaches the image object from the view

If the image object was inserted into the view, property changes will be reflected in the view. To disable this feature, 'detach" can be called after which the image object becomes inactive and changes will no longer be reflected in the view.

This method has been introduced in version 0.25.

dup

Signature: *[const]* new Image ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

filename

Signature: *[const]* string **filename**

Description: Gets the name of the file loaded of an empty string if not file is loaded

Returns: The file name (path)

from_s

Signature: *[static]* new Image ptr **from_s** (string s)

Description: Creates an image from the string returned by to_s.

This method has been introduced in version 0.27.

get_pixel

(1) Signature: *[const]* double **get_pixel** (unsigned long x, unsigned long y)

Description: Gets one pixel (monochrome only)

x: The x coordinate of the pixel (0..width()-1)

y: The y coordinate of the pixel (mathematical order: 0 is the lowest, 0..height()-1)

If x or y value exceeds the image bounds, this method returns 0.0. This method is valid for monochrome images only. For color images it will return 0.0 always. Use is_color? to decide whether the image is a color image or monochrome one.

(2) Signature: *[const]* double **get_pixel** (unsigned long x, unsigned long y, unsigned int component)

Description: Gets one pixel (monochrome and color)

x: The x coordinate of the pixel (0..width()-1)

y: The y coordinate of the pixel (mathematical order: 0 is the lowest, 0..height()-1)

component: 0 for red, 1 for green, 2 for blue.

If the component index, x or y value exceeds the image bounds, this method returns 0.0. For monochrome images, the component index is ignored.

height

Signature: *[const]* unsigned long **height**

Description: Gets the height of the image in pixels

Returns: The height in pixels

id

Signature: *[const]* unsigned long **id**

Description: Gets the Id

The Id is an arbitrary integer that can be used to track the evolution of an image object. The Id is not changed when the object is edited. On initialization, a unique Id is given to the object. The Id cannot be changed. This behaviour has been modified in version 0.20.

is_color?

Signature: *[const]* bool **is_color?**

Description: Returns true, if the image is a color image

Returns: True, if the image is a color image

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

is_empty?

Signature: *[const]* bool **is_empty?**

Description: Returns true, if the image does not contain any data (i.e. is default constructed)

Returns: True, if the image is empty

is_valid?

Signature: *[const]* bool **is_valid?**

Description: Returns a value indicating whether the object is a valid reference.

If this value is true, the object represents an image on the screen. Otherwise, the object is a 'detached' image which does not have a representation on the screen.

This method was introduced in version 0.25.

is_visible?

Signature: *[const]* bool **is_visible?**

Description: Gets a flag indicating whether the image object is visible

An image object can be made invisible by setting the visible property to false.

This method has been introduced in version 0.20.

Python specific notes:
The object exposes a readable attribute 'visible'. This is the getter.

mask

Signature: *[const]* bool **mask** (unsigned long x, unsigned long y)

Description: Gets the mask for one pixel

x:	The x coordinate of the pixel (0..width()-1)
y:	The y coordinate of the pixel (mathematical order: 0 is the lowest, 0..height()-1)
Returns:	false if the pixel is not drawn.

See set_mask for details about the mask.

This method has been introduced in version 0.23.

mask_data

Signature: bool[] **mask_data**

Description: Gets the mask from a array of boolean values

See set_mask_data for a description of the data field.

This method has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'mask_data'. This is the getter.

mask_data=

Signature: void **mask_data=** (bool[] mask_data)

Description: Sets the mask from a array of boolean values

The order of the boolean values is line first, from bottom to top and left to right and is the same as the order in the data array.

This method has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'mask_data'. This is the setter.

matrix

Signature: *[const]* Matrix3d **matrix**

Description: Returns the pixel-to-micron transformation matrix

This transformation matrix converts pixel coordinates (0,0 being the center and each pixel having the dimension of pixel_width and pixel_height) to micron coordinates. The coordinate of the pixel is the lower left corner of the pixel.

The matrix is more general than the transformation used before and supports shear and perspective transformation. This property replaces the trans property which is still functional, but deprecated.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a readable attribute 'matrix'. This is the getter.

matrix=

Signature: void **matrix=** (const Matrix3d t)

Description: Sets the transformation matrix

This transformation matrix converts pixel coordinates (0,0 being the center and each pixel having the dimension of pixel_width and pixel_height) to micron coordinates. The coordinate of the pixel is the lower left corner of the pixel.

The matrix is more general than the transformation used before and supports shear and perspective transformation. This property replaces the trans property which is still functional, but deprecated.

This method has been introduced in version 0.22.

Python specific notes:
The object exposes a writable attribute 'matrix'. This is the setter.

max_value

Signature: *[const]* double **max_value**

Description: Sets the maximum value

See the max_value method for the description of the maximum value property.

Python specific notes:
The object exposes a readable attribute 'max_value'. This is the getter.

max_value=

Signature: void **max_value=** (double v)

Description: Gets the upper limit of the values in the data set

This value determines the upper end of the data mapping (i.e. white value etc.). It does not necessarily correspond to the maximum value of the data set but it must be larger than that.

Python specific notes:
The object exposes a writable attribute 'max_value'. This is the setter.

min_value

Signature: *[const]* double **min_value**

Description: Gets the upper limit of the values in the data set

This value determines the upper end of the data mapping (i.e. white value etc.). It does not necessarily correspond to the minimum value of the data set but it must be larger than that.

Python specific notes:
The object exposes a readable attribute 'min_value'. This is the getter.

min_value=

Signature: void **min_value=** (double v)

Description: Sets the minimum value

See min_value for the description of the minimum value property.

Python specific notes:
The object exposes a writable attribute 'min_value'. This is the setter.

new

(1) Signature: *[static]* new Image ptr **new**

Description: Create a new image with the default attributes

This will create an empty image without data and no particular pixel width or related. Use the read_file or set_data methods to set image properties and pixel values.

Python specific notes:

This method is the default initializer of the object.

(2) Signature: *[static]* new Image ptr **new** (string filename, const DCplxTrans trans = unity)

Description: Constructor from a image file

filename:	The path to the image file to load.
trans:	The transformation to apply to the image when displaying it.

This constructor creates an image object from a file (which can have any format supported by Qt) and a transformation. The image will originally be put to position 0,0 (lower left corner) and each pixel will have a size of 1. The transformation describes how to transform this image into micron space.

Python specific notes:
This method is the default initializer of the object.

(3) Signature: *[static]* new Image ptr **new** (const PixelBuffer pixels, const DCplxTrans trans = unity)

Description: Constructor from a image pixel buffer

filename:	The path to the image file to load.
trans:	The transformation to apply to the image when displaying it.

This constructor creates an image object from a pixel buffer object. This object holds RGB or mono image data similar to QImage, except it is available also when Qt is not available (e.g. inside the Python module).

The image will originally be put to position 0,0 (lower left corner) and each pixel will have a size of 1. The transformation describes how to transform this image into micron space.

Python specific notes:
This method is the default initializer of the object.

(4) Signature: *[static]* new Image ptr **new** (const QImage image, const DCplxTrans trans = unity)

Description: Constructor from a image pixel buffer

filename:	The path to the image file to load.
trans:	The transformation to apply to the image when displaying it.

This constructor creates an image object from a pixel QImage object and uses RGB or mono image data to generate the image.

The image will originally be put to position 0,0 (lower left corner) and each pixel will have a size of 1. The transformation describes how to transform this image into micron space.

Python specific notes:
This method is the default initializer of the object.

(5) Signature: *[static]* new Image ptr **new** (unsigned long w, unsigned long h, double[] data)

Description: Constructor for a monochrome image with the given pixel values

w:	The width of the image
h:	The height of the image
d:	The data (see method description)

This constructor creates an image from the given pixel values. The values have to be organized line by line. Each line must consist of "w" values where the first value is the leftmost pixel. Note, that the rows are oriented in the mathematical sense (first one is the lowest) contrary to the common convention for image data. Initially the pixel width and height will be 1 micron and the data range will be 0 to 1.0 (black to white level). To adjust the data range use the min_value and max_value properties.

Python specific notes:

This method is the default initializer of the object.

(6) Signature: *[static]* new Image ptr **new** (unsigned long w, unsigned long h, const DCplxTrans trans, double[] data)

Description: Constructor for a monochrome image with the given pixel values

w:	The width of the image
h:	The height of the image
trans:	The transformation from pixel space to micron space
d:	The data (see method description)

This constructor creates an image from the given pixel values. The values have to be organized line by line. Each line must consist of "w" values where the first value is the leftmost pixel. Note, that the rows are oriented in the mathematical sense (first one is the lowest) contrary to the common convention for image data. Initially the pixel width and height will be 1 micron and the data range will be 0 to 1.0 (black to white level). To adjust the data range use the min_value and max_value properties.

Python specific notes:
This method is the default initializer of the object.

(7) Signature: *[static]* new Image ptr **new** (unsigned long w, unsigned long h, double[] red, double[] green, double[] blue)

Description: Constructor for a color image with the given pixel values

w:	The width of the image
h:	The height of the image
red:	The red channel data set which will become owned by the image
green:	The green channel data set which will become owned by the image
blue:	The blue channel data set which will become owned by the image

This constructor creates an image from the given pixel values. The values have to be organized line by line and separated by color channel. Each line must consist of "w" values where the first value is the leftmost pixel. Note, that the rows are oriented in the mathematical sense (first one is the lowest) contrary to the common convention for image data. Initially the pixel width and height will be 1 micron and the data range will be 0 to 1.0 (black to white level). To adjust the data range use the min_value and max_value properties.

Python specific notes:
This method is the default initializer of the object.

(8) Signature: *[static]* new Image ptr **new** (unsigned long w, unsigned long h, const DCplxTrans trans, double[] red, double[] green, double[] blue)

Description: Constructor for a color image with the given pixel values

w:	The width of the image
h:	The height of the image
trans:	The transformation from pixel space to micron space
red:	The red channel data set which will become owned by the image
green:	The green channel data set which will become owned by the image
blue:	The blue channel data set which will become owned by the image

This constructor creates an image from the given pixel values. The values have to be organized line by line and separated by color channel. Each line must consist of "w" values where the first value is

the leftmost pixel. Note, that the rows are oriented in the mathematical sense (first one is the lowest) contrary to the common convention for image data. Initially the pixel width and height will be 1 micron and the data range will be 0 to 1.0 (black to white level). To adjust the data range use the min_value and max_value properties.

Python specific notes:
This method is the default initializer of the object.

pixel_height

Signature: *[const]* double **pixel_height**

Description: Gets the pixel height

See pixel_height= for a description of that property.

Starting with version 0.22, this property is incorporated into the transformation matrix. This property is provided for convenience only.

Python specific notes:
The object exposes a readable attribute 'pixel_height'. This is the getter.

pixel_height=

Signature: void **pixel_height=** (double h)

Description: Sets the pixel height

The pixel height determines the height of on pixel in the original space which is transformed to micron space with the transformation.

Starting with version 0.22, this property is incorporated into the transformation matrix. This property is provided for convenience only.

Python specific notes:
The object exposes a writable attribute 'pixel_height'. This is the setter.

pixel_width

Signature: *[const]* double **pixel_width**

Description: Gets the pixel width

See pixel_width= for a description of that property.

Starting with version 0.22, this property is incorporated into the transformation matrix. This property is provided for convenience only.

Python specific notes:
The object exposes a readable attribute 'pixel_width'. This is the getter.

pixel_width=

Signature: void **pixel_width=** (double w)

Description: Sets the pixel width

The pixel width determines the width of on pixel in the original space which is transformed to micron space with the transformation.

Starting with version 0.22, this property is incorporated into the transformation matrix. This property is provided for convenience only.

Python specific notes:
The object exposes a writable attribute 'pixel_width'. This is the setter.

read

Signature: *[static]* new Image ptr **read** (string path)

Description: Loads the image from the given path.

This method expects the image file as a KLayout image format file (.lyimg). This is a XML-based format containing the image data plus placement and transformation information for the image placement. In addition, image manipulation parameters for false color display and color channel enhancement are embedded.

This method has been introduced in version 0.27.

set_data

(1) Signature: void **set_data** (unsigned long w, unsigned long h, double[] d)

Description: Writes the image data field (monochrome)

- **w**: The width of the new data
- **h**: The height of the new data
- **d**: The (monochrome) data to load into the image

See the constructor description for the data organisation in that field.

(2) Signature: void **set_data** (unsigned long w, unsigned long h, double[] r, double[] g, double[] b)

Description: Writes the image data field (color)

- **w**: The width of the new data
- **h**: The height of the new data
- **r**: The red channel data to load into the image
- **g**: The green channel data to load into the image
- **b**: The blue channel data to load into the image

See the constructor description for the data organisation in that field.

set_mask

Signature: void **set_mask** (unsigned long x, unsigned long y, bool m)

Description: Sets the mask for a pixel

- **x**: The x coordinate of the pixel (0..width()-1)
- **y**: The y coordinate of the pixel (mathematical order: 0 is the lowest, 0..height()-1)
- **m**: The mask

If the mask of a pixel is set to false, the pixel is not drawn. The default is true for all pixels.

This method has been introduced in version 0.23.

set_pixel

(1) Signature: void **set_pixel** (unsigned long x, unsigned long y, double v)

Description: Sets one pixel (monochrome)

- **x**: The x coordinate of the pixel (0..width()-1)
- **y**: The y coordinate of the pixel (mathematical order: 0 is the lowest, 0..height()-1)
- **v**: The value

If the component index, x or y value exceeds the image bounds of the image is a color image, this method does nothing.

(2) Signature: void **set_pixel** (unsigned long x, unsigned long y, double r, double g, double b)

Description: Sets one pixel (color)

- **x**: The x coordinate of the pixel (0..width()-1)
- **y**: The y coordinate of the pixel (mathematical order: 0 is the lowest, 0..height()-1)
- **red**: The red component
- **green**: The green component

blue: The blue component

If the component index, x or y value exceeds the image bounds of the image is not a color image, this method does nothing.

to_s

Signature: *[const]* string **to_s**

Description: Converts the image to a string

Returns: The string

The string returned can be used to create an image object using from_s.

Python specific notes:
This method is also available as 'str(object)'.

trans

Signature: *[const]* DCplxTrans **trans**

Description: Returns the pixel-to-micron transformation

This transformation converts pixel coordinates (0,0 being the lower left corner and each pixel having the dimension of pixel_width and pixel_height) to micron coordinates. The coordinate of the pixel is the lower left corner of the pixel.

The general property is matrix which also allows perspective and shear transformation. This property will only work, if the transformation does not include perspective or shear components. Therefore this property is deprecated. Please note that for backward compatibility, the rotation center is pixel 0,0 (lowest left one), while it is the image center for the matrix transformation.

Python specific notes:
The object exposes a readable attribute 'trans'. This is the getter.

trans=

Signature: void **trans=** (const DCplxTrans t)

Description: Sets the transformation

This transformation converts pixel coordinates (0,0 being the lower left corner and each pixel having the dimension of pixel_width and pixel_height) to micron coordinates. The coordinate of the pixel is the lower left corner of the pixel.

The general property is matrix which also allows perspective and shear transformation. Please note that for backward compatibility, the rotation center is pixel 0,0 (lowest left one), while it is the image center for the matrix transformation.

Python specific notes:
The object exposes a writable attribute 'trans'. This is the setter.

transformed

(1) Signature: *[const]* Image **transformed** (const DTrans t)

Description: Transforms the image with the given simple transformation

t: The transformation to apply

Returns: The transformed object

(2) Signature: *[const]* Image **transformed** (const Matrix3d t)

Description: Transforms the image with the given matrix transformation

t: The transformation to apply (a matrix)

Returns: The transformed object

This method has been introduced in version 0.22.

(3) Signature: *[const]* Image **transformed** (const DCplxTrans t)

Description: Transforms the image with the given complex transformation

t: The magnifying transformation to apply

Returns: The transformed object

transformed_cplx

Signature: *[const]* Image **transformed_cplx** (const DCplxTrans t)

Description: Transforms the image with the given complex transformation

t: The magnifying transformation to apply

Returns: The transformed object

Use of this method is deprecated. Use transformed instead

transformed_matrix

Signature: *[const]* Image **transformed_matrix** (const Matrix3d t)

Description: Transforms the image with the given matrix transformation

t: The transformation to apply (a matrix)

Returns: The transformed object

Use of this method is deprecated. Use transformed instead

This method has been introduced in version 0.22.

update

Signature: void **update**

Description: Forces an update of the view

Usually it is not required to call this method. The image object is automatically synchronized with the view's image objects. For performance reasons this update is delayed to collect multiple update requests. Calling 'update' will ensure immediate updates.

This method has been introduced in version 0.25.

visible=

Signature: void **visible=** (bool v)

Description: Sets the visibility

See the is_visible? method for a description of this property.

This method has been introduced in version 0.20.

Python specific notes:
The object exposes a writable attribute 'visible'. This is the setter.

width

Signature: *[const]* unsigned long **width**

Description: Gets the width of the image in pixels

Returns: The width in pixels

write

Signature: *[const]* void **write** (string path)

Description: Saves the image to KLayout's image format (.lyimg)

This method has been introduced in version 0.27.

z_position

Signature: *[const]* int **z_position**

Description: Gets the z position of the image

Images with a higher z position are painted in front of images with lower z position. The z value is an integer that controls the position relative to other images.

This method was introduced in version 0.25.

Python specific notes:
The object exposes a readable attribute 'z_position'. This is the getter.

z_position=

Signature: void **z_position=** (int z)

Description: Sets the z position of the image

See z_position for details about the z position attribute.

This method was introduced in version 0.25.

Python specific notes:
The object exposes a writable attribute 'z_position'. This is the setter.

4.230. API reference - Class HelpDialog

Notation used in Ruby API documentation

Module: lay

Description: The help dialog

Class hierarchy: HelpDialog » QDialog » QWidget » QObject

This class makes the help dialog available as an individual object.

This class has been added in version 0.25.

Public constructors

HelpDialog ptr	**new**	(bool modal)	Creates a new help dialog
HelpDialog ptr	**new**	(QWidget ptr parent, bool modal)	Creates a new help dialog

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**load**	(string url)	Loads the specified URL
	void	**search**	(string topic)	Issues a search on the specified topic

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

load

Signature: void **load** (string url)

Description: Loads the specified URL

This method will call the page with the given URL.

new

(1) Signature: *[static]* HelpDialog ptr **new** (bool modal)

Description: Creates a new help dialog

If the modal flag is true, the dialog will be shown as a modal window.

(2) Signature: *[static]* HelpDialog ptr **new** (QWidget ptr parent, bool modal)

Description: Creates a new help dialog

If the modal flag is true, the dialog will be shown as a modal window.

search

Signature: void **search** (string topic)

Description: Issues a search on the specified topic

This method will call the search page with the given topic.

4.231. API reference - Class HelpSource

Notation used in Ruby API documentation

Module: lay

Description: A BrowserSource implementation delivering the help text for the help dialog

Class hierarchy: HelpSource » BrowserSource

This class can be used together with a BrowserPanel or BrowserDialog object to implement custom help systems.

The basic URL's served by this class are: "int:/index.xml" for the index page and "int:/search.xml?string=..." for the search topic retrieval.

This class has been added in version 0.25.

Public methods

	void	**_assign**	(const HelpSource other)	Assigns another object to self
	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	new HelpSource ptr	**_dup**		Creates a copy of self
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	QDomDocument	**get_dom**	(string path)	Reserved for internal use
[const]	variant	**get_option**	(string key)	Reserved for internal use
	string	**parent_of**	(string path)	Reserved internal use
	void	**scan**		Reserved internal use
	void	**set_option**	(string key, variant value)	Reserved for internal use
	string	**title_for**	(string path)	Reserved internal use
	string[]	**urls**		Reserved for internal use

Public static methods and constants

void	**create_index_file**	(string path)	Reserved internal use
new HelpSource ptr	**plain**		Reserved for internal use

Detailed description

_assign

Signature: void **_assign** (const HelpSource other)

Description: Assigns another object to self

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_dup

Signature: *[const]* new HelpSource ptr **_dup**

Description: Creates a copy of self

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

create_index_file	**Signature**: *[static]* void **create_index_file** (string path) **Description**: Reserved internal use
get_dom	**Signature**: QDomDocument **get_dom** (string path) **Description**: Reserved for internal use
get_option	**Signature**: *[const]* variant **get_option** (string key) **Description**: Reserved for internal use
parent_of	**Signature**: string **parent_of** (string path) **Description**: Reserved internal use
plain	**Signature**: *[static]* new HelpSource ptr **plain** **Description**: Reserved for internal use
scan	**Signature**: void **scan** **Description**: Reserved internal use
set_option	**Signature**: void **set_option** (string key, variant value) **Description**: Reserved for internal use
title_for	**Signature**: string **title_for** (string path) **Description**: Reserved internal use
urls	**Signature**: string[] **urls** **Description**: Reserved for internal use

4.232. API reference - Class MainWindow

Notation used in Ruby API documentation

Module: lay

Description: The main application window and central controller object

Class hierarchy: MainWindow » QMainWindow » QWidget » QObject

This object first is the main window but also the main controller. The main controller is the port by which access can be gained to all the data objects, view and other aspects of the program.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**call_menu**	(string symbol)	Calls the menu item with the provided symbol.
	void	**cancel**		Cancels current editing operations
	void	**clear_config**		Clears the configuration parameters
	void	**clone_current_view**		Clones the current view and make it current
	void	**close_all**		Closes all views
	void	**close_current_view**		Closes the current view
	void	**commit_config**		Commits the configuration settings
	CellView	**create_layout**	(int mode)	Creates a new, empty layout
	CellView	**create_layout**	(string tech, int mode)	Creates a new, empty layout with the given technology
	int	**create_view**		Creates a new, empty view
	LayoutView ptr	**current_view**		Returns a reference to the current view's object
[const]	int	**current_view_index**		Returns the current view's index
	void	**current_view_index=**	(int index)	Selects the view with the given index

[const]	Dispatcher ptr	**dispatcher**		Gets the dispatcher interface (the plugin root configuration space)
	void	**exit**		Schedules an exit for the application
	variant	**get_config**	(string name)	Gets the value of a local configuration parameter
	string[]	**get_config_names**		Gets the configuration parameter names
	map<string,string>	**get_default_key_bindings**		Gets the default key bindings
	map<string,bool>	**get_default_menu_items_h**		Gets the flags indicating whether menu items are hidden by default
	map<string,string>	**get_key_bindings**		Gets the current key bindings
	map<string,bool>	**get_menu_items_hidden**		Gets the flags indicating whether menu items are hidden
[const]	double	**grid_micron**		Gets the global grid in micron
[const]	int	**index_of**	(const LayoutView ptr view)	Gets the index of the given view
	string	**initial_technology**		Gets the technology used for creating or loading layouts (unless explicitly specified)
	void	**initial_technology=**	(string tech)	Sets the technology used for creating or loading layouts (unless explicitly specified)
	CellView	**load_layout**	(string filename, int mode = 1)	Loads a new layout
	CellView	**load_layout**	(string filename, string tech, int mode = 1)	Loads a new layout and associate it with the given technology
	CellView	**load_layout**	(string filename, const LoadLayoutOptions options, int mode = 1)	Loads a new layout with the given options
	CellView	**load_layout**	(string filename, const LoadLayoutOptions options, string tech, int mode = 1)	Loads a new layout with the given options and associate it with the given technology
	Manager	**manager**		Gets the Manager object of this window
	AbstractMenu ptr	**menu**		Returns a reference to the abstract menu
	void	**message**	(string message, int time = infinite)	Displays a message in the status bar

[signal]	void	on_current_view_changed		An event indicating that the current view has changed
[signal]	void	on_session_about_to_be_restored		An event indicating that a session is about to be restored
[signal]	void	on_session_restored		An event indicating that a session was restored
[signal]	void	on_view_closed	(int index)	An event indicating that a view was closed
[signal]	void	on_view_created	(int index)	An event indicating that a new view was created
	bool	read_config	(string file_name)	Reads the configuration from a file
	void	redraw		Redraws the current view
	void	resize	(int width, int height)	Resizes the window
	void	restore_session	(string fn)	Restores a session from the given file
	void	save_session	(string fn)	Saves the session to the given file
	void	set_config	(string name, string value)	Set a local configuration parameter with the given name to the given value
	void	set_key_bindings	(map<string,string> bindings)	Sets key bindings.
	void	set_menu_items_hidden	(map<string,bool> flags)	sets the flags indicating whether menu items are hidden
	void	show_macro_editor	(string cat = , bool add = false)	Shows the macro editor
[const]	bool	synchronous		Gets a value indicating whether synchronous mode is activated
	void	synchronous=	(bool sync_mode)	Puts the main window into synchronous mode
[const]	string	title		Gets the window title
	void	title=	(string title)	Sets the window title
	LayoutView ptr	view	(int n)	Returns a reference to a view object by index
[const]	unsigned int	views		Returns the number of views
	bool	write_config	(string file_name)	Writes configuration to a file

Public static methods and constants

MainWindow ptr	**instance**	Gets application's main window instance
string[]	**menu_symbols**	Gets all available menu symbols (see call_menu).

Deprecated methods (protected, public, static, non-static and constructors)

void	**cm_adjust_origin**	Use of this method is deprecated
void	**cm_bookmark_view**	Use of this method is deprecated
void	**cm_cancel**	Use of this method is deprecated
void	**cm_cell_copy**	Use of this method is deprecated
void	**cm_cell_cut**	Use of this method is deprecated
void	**cm_cell_delete**	Use of this method is deprecated
void	**cm_cell_flatten**	Use of this method is deprecated
void	**cm_cell_hide**	Use of this method is deprecated
void	**cm_cell_paste**	Use of this method is deprecated
void	**cm_cell_rename**	Use of this method is deprecated
void	**cm_cell_select**	Use of this method is deprecated
void	**cm_cell_show**	Use of this method is deprecated
void	**cm_cell_show_all**	Use of this method is deprecated
void	**cm_clear_layer**	Use of this method is deprecated
void	**cm_clone**	Use of this method is deprecated
void	**cm_close**	Use of this method is deprecated
void	**cm_close_all**	Use of this method is deprecated
void	**cm_copy**	Use of this method is deprecated
void	**cm_copy_layer**	Use of this method is deprecated
void	**cm_cut**	Use of this method is deprecated
void	**cm_dec_max_hier**	Use of this method is deprecated
void	**cm_delete**	Use of this method is deprecated
void	**cm_delete_layer**	Use of this method is deprecated
void	**cm_edit_layer**	Use of this method is deprecated

void	cm_exit	Use of this method is deprecated
void	cm_goto_position	Use of this method is deprecated
void	cm_help_about	Use of this method is deprecated
void	cm_inc_max_hier	Use of this method is deprecated
void	cm_last_display_state	Use of this method is deprecated
void	cm_layout_props	Use of this method is deprecated
void	cm_load_bookmarks	Use of this method is deprecated
void	cm_load_layer_props	Use of this method is deprecated
void	cm_lv_add_missing	Use of this method is deprecated
void	cm_lv_delete	Use of this method is deprecated
void	cm_lv_expand_all	Use of this method is deprecated
void	cm_lv_group	Use of this method is deprecated
void	cm_lv_hide	Use of this method is deprecated
void	cm_lv_hide_all	Use of this method is deprecated
void	cm_lv_insert	Use of this method is deprecated
void	cm_lv_new_tab	Use of this method is deprecated
void	cm_lv_regroup_by_datatype	Use of this method is deprecated
void	cm_lv_regroup_by_index	Use of this method is deprecated
void	cm_lv_regroup_by_layer	Use of this method is deprecated
void	cm_lv_regroup_flatten	Use of this method is deprecated
void	cm_lv_remove_tab	Use of this method is deprecated
void	cm_lv_remove_unused	Use of this method is deprecated
void	cm_lv_rename	Use of this method is deprecated
void	cm_lv_rename_tab	Use of this method is deprecated
void	cm_lv_select_all	Use of this method is deprecated
void	cm_lv_show	Use of this method is deprecated
void	cm_lv_show_all	Use of this method is deprecated
void	cm_lv_show_only	Use of this method is deprecated
void	cm_lv_sort_by_dli	Use of this method is deprecated

void	cm_lv_sort_by_idl	Use of this method is deprecated
void	cm_lv_sort_by_ild	Use of this method is deprecated
void	cm_lv_sort_by_ldi	Use of this method is deprecated
void	cm_lv_sort_by_name	Use of this method is deprecated
void	cm_lv_source	Use of this method is deprecated
void	cm_lv_ungroup	Use of this method is deprecated
void	cm_macro_editor	Use of this method is deprecated
void	cm_manage_bookmarks	Use of this method is deprecated
void	cm_max_hier	Use of this method is deprecated
void	cm_max_hier_0	Use of this method is deprecated
void	cm_max_hier_1	Use of this method is deprecated
void	cm_navigator_close	Use of this method is deprecated
void	cm_new_cell	Use of this method is deprecated
void	cm_new_layer	Use of this method is deprecated
void	cm_new_layout	Use of this method is deprecated
void	cm_new_panel	Use of this method is deprecated
void	cm_next_display_state	Use of this method is deprecated
void	cm_open	Use of this method is deprecated
void	cm_open_current_cell	Use of this method is deprecated
void	cm_open_new_view	Use of this method is deprecated
void	cm_open_too	Use of this method is deprecated
void	cm_packages	Use of this method is deprecated
void	cm_pan_down	Use of this method is deprecated
void	cm_pan_left	Use of this method is deprecated
void	cm_pan_right	Use of this method is deprecated
void	cm_pan_up	Use of this method is deprecated
void	cm_paste	Use of this method is deprecated
void	cm_prev_display_state	Use of this method is deprecated
void	cm_print	Use of this method is deprecated

void	cm_pull_in	Use of this method is deprecated
void	cm_reader_options	Use of this method is deprecated
void	cm_redo	Use of this method is deprecated
void	cm_redraw	Use of this method is deprecated
void	cm_reload	Use of this method is deprecated
void	cm_reset_window_state	Use of this method is deprecated
void	cm_restore_session	Use of this method is deprecated
void	cm_save	Use of this method is deprecated
void	cm_save_all	Use of this method is deprecated
void	cm_save_as	Use of this method is deprecated
void	cm_save_bookmarks	Use of this method is deprecated
void	cm_save_current_cell_as	Use of this method is deprecated
void	cm_save_layer_props	Use of this method is deprecated
void	cm_save_session	Use of this method is deprecated
void	cm_screenshot	Use of this method is deprecated
void	cm_screenshot_to_clipboard	Use of this method is deprecated
void	cm_sel_flip_x	Use of this method is deprecated
void	cm_sel_flip_y	Use of this method is deprecated
void	cm_sel_free_rot	Use of this method is deprecated
void	cm_sel_move	Use of this method is deprecated
void	cm_sel_move_to	Use of this method is deprecated
void	cm_sel_rot_ccw	Use of this method is deprecated
void	cm_sel_rot_cw	Use of this method is deprecated
void	cm_sel_scale	Use of this method is deprecated
void	cm_select_all	Use of this method is deprecated
void	cm_select_cell	Use of this method is deprecated
void	cm_select_current_cell	Use of this method is deprecated
void	cm_setup	Use of this method is deprecated
void	cm_show_properties	Use of this method is deprecated

void	cm_technologies		Use of this method is deprecated
void	cm_undo		Use of this method is deprecated
void	cm_unselect_all		Use of this method is deprecated
void	cm_view_log		Use of this method is deprecated
void	cm_zoom_fit		Use of this method is deprecated
void	cm_zoom_fit_sel		Use of this method is deprecated
void	cm_zoom_in		Use of this method is deprecated
void	cm_zoom_out		Use of this method is deprecated
void	enable_edits	(bool enable)	Use of this method is deprecated
void	select_view	(int index)	Use of this method is deprecated. Use current_view_index= instead
void	synchronous	(bool sync_mode)	Use of this method is deprecated. Use synchronous= instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

	After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
call_menu	**Signature**: void **call_menu** (string symbol) **Description**: Calls the menu item with the provided symbol. To obtain all symbols, use menu_symbols. This method has been introduced in version 0.27 and replaces the previous cm_... methods. Instead of calling a specific cm_... method, use LayoutView#call_menu with 'cm_...' as the symbol.
cancel	**Signature**: void **cancel** **Description**: Cancels current editing operations This method call cancels all current editing operations and restores normal mouse mode.
clear_config	**Signature**: void **clear_config** **Description**: Clears the configuration parameters This method is provided for using MainWindow without an Application object. It's a convience method which is equivalent to 'dispatcher().clear_config()'. See Dispatcher#clear_config for details. This method has been introduced in version 0.27.
clone_current_view	**Signature**: void **clone_current_view** **Description**: Clones the current view and make it current
close_all	**Signature**: void **close_all** **Description**: Closes all views This method unconditionally closes all views. No dialog will be opened if unsaved edits exist. This method was added in version 0.18.
close_current_view	**Signature**: void **close_current_view** **Description**: Closes the current view This method does not open a dialog to ask which cell view to close if multiple cells are opened in the view, but rather closes all cells.

cm_adjust_origin	**Signature**: void **cm_adjust_origin** **Description**: 'cm_adjust_origin' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_adjust_origin')" instead.
cm_bookmark_view	**Signature**: void **cm_bookmark_view** **Description**: 'cm_bookmark_view' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_bookmark_view')" instead.
cm_cancel	**Signature**: void **cm_cancel** **Description**: 'cm_cancel' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_cancel')" instead.
cm_cell_copy	**Signature**: void **cm_cell_copy** **Description**: 'cm_cell_copy' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_cell_copy')" instead.
cm_cell_cut	**Signature**: void **cm_cell_cut** **Description**: 'cm_cell_cut' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_cell_cut')" instead.
cm_cell_delete	**Signature**: void **cm_cell_delete** **Description**: 'cm_cell_delete' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_cell_delete')" instead.
cm_cell_flatten	**Signature**: void **cm_cell_flatten** **Description**: 'cm_cell_flatten' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_cell_flatten')" instead.
cm_cell_hide	**Signature**: void **cm_cell_hide** **Description**: 'cm_cell_hide' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_cell_hide')" instead.
cm_cell_paste	**Signature**: void **cm_cell_paste** **Description**: 'cm_cell_paste' action. Use of this method is deprecated

	This method is deprecated in version 0.27. Use "call_menu('cm_cell_paste')" instead.
cm_cell_rename	**Signature**: void **cm_cell_rename** **Description**: 'cm_cell_rename' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_cell_rename')" instead.
cm_cell_select	**Signature**: void **cm_cell_select** **Description**: 'cm_cell_select' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_cell_select')" instead.
cm_cell_show	**Signature**: void **cm_cell_show** **Description**: 'cm_cell_show' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_cell_show')" instead.
cm_cell_show_all	**Signature**: void **cm_cell_show_all** **Description**: 'cm_cell_show_all' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_cell_show_all')" instead.
cm_clear_layer	**Signature**: void **cm_clear_layer** **Description**: 'cm_clear_layer' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_clear_layer')" instead.
cm_clone	**Signature**: void **cm_clone** **Description**: 'cm_clone' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_clone')" instead.
cm_close	**Signature**: void **cm_close** **Description**: 'cm_close' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_close')" instead.
cm_close_all	**Signature**: void **cm_close_all** **Description**: 'cm_close_all' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_close_all')" instead.
cm_copy	**Signature**: void **cm_copy** **Description**: 'cm_copy' action.

	Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_copy')" instead.
cm_copy_layer	**Signature**: void **cm_copy_layer** **Description**: 'cm_copy_layer' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_copy_layer')" instead.
cm_cut	**Signature**: void **cm_cut** **Description**: 'cm_cut' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_cut')" instead.
cm_dec_max_hier	**Signature**: void **cm_dec_max_hier** **Description**: 'cm_dec_max_hier' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_dec_max_hier')" instead.
cm_delete	**Signature**: void **cm_delete** **Description**: 'cm_delete' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_delete')" instead.
cm_delete_layer	**Signature**: void **cm_delete_layer** **Description**: 'cm_delete_layer' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_delete_layer')" instead.
cm_edit_layer	**Signature**: void **cm_edit_layer** **Description**: 'cm_edit_layer' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_edit_layer')" instead.
cm_exit	**Signature**: void **cm_exit** **Description**: 'cm_exit' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_exit')" instead.
cm_goto_position	**Signature**: void **cm_goto_position** **Description**: 'cm_goto_position' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_goto_position')" instead.

cm_help_about	**Signature**: void **cm_help_about** **Description**: 'cm_help_about' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_help_about')" instead.
cm_inc_max_hier	**Signature**: void **cm_inc_max_hier** **Description**: 'cm_inc_max_hier' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_inc_max_hier')" instead.
cm_last_display_state	**Signature**: void **cm_last_display_state** **Description**: 'cm_prev_display_state|#cm_last_display_state' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_prev_display_state|#cm_last_display_state')" instead.
cm_layout_props	**Signature**: void **cm_layout_props** **Description**: 'cm_layout_props' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_layout_props')" instead.
cm_load_bookmarks	**Signature**: void **cm_load_bookmarks** **Description**: 'cm_load_bookmarks' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_load_bookmarks')" instead.
cm_load_layer_props	**Signature**: void **cm_load_layer_props** **Description**: 'cm_load_layer_props' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_load_layer_props')" instead.
cm_lv_add_missing	**Signature**: void **cm_lv_add_missing** **Description**: 'cm_lv_add_missing' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_add_missing')" instead.
cm_lv_delete	**Signature**: void **cm_lv_delete** **Description**: 'cm_lv_delete' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_delete')" instead.
cm_lv_expand_all	**Signature**: void **cm_lv_expand_all** **Description**: 'cm_lv_expand_all' action.

	Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_expand_all')" instead.
cm_lv_group	**Signature**: void **cm_lv_group** **Description**: 'cm_lv_group' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_group')" instead.
cm_lv_hide	**Signature**: void **cm_lv_hide** **Description**: 'cm_lv_hide' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_hide')" instead.
cm_lv_hide_all	**Signature**: void **cm_lv_hide_all** **Description**: 'cm_lv_hide_all' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_hide_all')" instead.
cm_lv_insert	**Signature**: void **cm_lv_insert** **Description**: 'cm_lv_insert' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_insert')" instead.
cm_lv_new_tab	**Signature**: void **cm_lv_new_tab** **Description**: 'cm_lv_new_tab' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_new_tab')" instead.
cm_lv_regroup_by_datatype	**Signature**: void **cm_lv_regroup_by_datatype** **Description**: 'cm_lv_regroup_by_datatype' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_regroup_by_datatype')" instead.
cm_lv_regroup_by_index	**Signature**: void **cm_lv_regroup_by_index** **Description**: 'cm_lv_regroup_by_index' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_regroup_by_index')" instead.
cm_lv_regroup_by_layer	**Signature**: void **cm_lv_regroup_by_layer** **Description**: 'cm_lv_regroup_by_layer' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_regroup_by_layer')" instead.

cm_lv_regroup_flatten	**Signature**: void **cm_lv_regroup_flatten** **Description**: 'cm_lv_regroup_flatten' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_regroup_flatten')" instead.
cm_lv_remove_tab	**Signature**: void **cm_lv_remove_tab** **Description**: 'cm_lv_remove_tab' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_remove_tab')" instead.
cm_lv_remove_unused	**Signature**: void **cm_lv_remove_unused** **Description**: 'cm_lv_remove_unused' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_remove_unused')" instead.
cm_lv_rename	**Signature**: void **cm_lv_rename** **Description**: 'cm_lv_rename' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_rename')" instead.
cm_lv_rename_tab	**Signature**: void **cm_lv_rename_tab** **Description**: 'cm_lv_rename_tab' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_rename_tab')" instead.
cm_lv_select_all	**Signature**: void **cm_lv_select_all** **Description**: 'cm_lv_select_all' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_select_all')" instead.
cm_lv_show	**Signature**: void **cm_lv_show** **Description**: 'cm_lv_show' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_show')" instead.
cm_lv_show_all	**Signature**: void **cm_lv_show_all** **Description**: 'cm_lv_show_all' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_show_all')" instead.
cm_lv_show_only	**Signature**: void **cm_lv_show_only** **Description**: 'cm_lv_show_only' action. Use of this method is deprecated

	This method is deprecated in version 0.27. Use "call_menu('cm_lv_show_only')" instead.
cm_lv_sort_by_dli	**Signature**: void **cm_lv_sort_by_dli** **Description**: 'cm_lv_sort_by_dli' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_sort_by_dli')" instead.
cm_lv_sort_by_idl	**Signature**: void **cm_lv_sort_by_idl** **Description**: 'cm_lv_sort_by_idl' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_sort_by_idl')" instead.
cm_lv_sort_by_ild	**Signature**: void **cm_lv_sort_by_ild** **Description**: 'cm_lv_sort_by_ild' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_sort_by_ild')" instead.
cm_lv_sort_by_ldi	**Signature**: void **cm_lv_sort_by_ldi** **Description**: 'cm_lv_sort_by_ldi' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_sort_by_ldi')" instead.
cm_lv_sort_by_name	**Signature**: void **cm_lv_sort_by_name** **Description**: 'cm_lv_sort_by_name' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_sort_by_name')" instead.
cm_lv_source	**Signature**: void **cm_lv_source** **Description**: 'cm_lv_source' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_source')" instead.
cm_lv_ungroup	**Signature**: void **cm_lv_ungroup** **Description**: 'cm_lv_ungroup' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_lv_ungroup')" instead.
cm_macro_editor	**Signature**: void **cm_macro_editor** **Description**: 'cm_macro_editor' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_macro_editor')" instead.
cm_manage_bookmarks	**Signature**: void **cm_manage_bookmarks** **Description**: 'cm_manage_bookmarks' action.

	Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_manage_bookmarks')" instead.
cm_max_hier	**Signature**: void **cm_max_hier** **Description**: 'cm_max_hier' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_max_hier')" instead.
cm_max_hier_0	**Signature**: void **cm_max_hier_0** **Description**: 'cm_max_hier_0' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_max_hier_0')" instead.
cm_max_hier_1	**Signature**: void **cm_max_hier_1** **Description**: 'cm_max_hier_1' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_max_hier_1')" instead.
cm_navigator_close	**Signature**: void **cm_navigator_close** **Description**: 'cm_navigator_close' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_navigator_close')" instead.
cm_new_cell	**Signature**: void **cm_new_cell** **Description**: 'cm_new_cell' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_new_cell')" instead.
cm_new_layer	**Signature**: void **cm_new_layer** **Description**: 'cm_new_layer' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_new_layer')" instead.
cm_new_layout	**Signature**: void **cm_new_layout** **Description**: 'cm_new_layout' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_new_layout')" instead.
cm_new_panel	**Signature**: void **cm_new_panel** **Description**: 'cm_new_panel' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_new_panel')" instead.

cm_next_display_state	**Signature**: void **cm_next_display_state** **Description**: 'cm_next_display_state' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_next_display_state')" instead.
cm_open	**Signature**: void **cm_open** **Description**: 'cm_open' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_open')" instead.
cm_open_current_cell	**Signature**: void **cm_open_current_cell** **Description**: 'cm_open_current_cell' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_open_current_cell')" instead.
cm_open_new_view	**Signature**: void **cm_open_new_view** **Description**: 'cm_open_new_view' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_open_new_view')" instead.
cm_open_too	**Signature**: void **cm_open_too** **Description**: 'cm_open_too' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_open_too')" instead.
cm_packages	**Signature**: void **cm_packages** **Description**: 'cm_packages' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_packages')" instead.
cm_pan_down	**Signature**: void **cm_pan_down** **Description**: 'cm_pan_down' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_pan_down')" instead.
cm_pan_left	**Signature**: void **cm_pan_left** **Description**: 'cm_pan_left' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_pan_left')" instead.
cm_pan_right	**Signature**: void **cm_pan_right** **Description**: 'cm_pan_right' action. Use of this method is deprecated

	This method is deprecated in version 0.27. Use "call_menu('cm_pan_right')" instead.
cm_pan_up	**Signature**: void **cm_pan_up** **Description**: 'cm_pan_up' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_pan_up')" instead.
cm_paste	**Signature**: void **cm_paste** **Description**: 'cm_paste' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_paste')" instead.
cm_prev_display_state	**Signature**: void **cm_prev_display_state** **Description**: 'cm_prev_display_state|#cm_last_display_state' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_prev_display_state|#cm_last_display_state')" instead.
cm_print	**Signature**: void **cm_print** **Description**: 'cm_print' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_print')" instead.
cm_pull_in	**Signature**: void **cm_pull_in** **Description**: 'cm_pull_in' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_pull_in')" instead.
cm_reader_options	**Signature**: void **cm_reader_options** **Description**: 'cm_reader_options' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_reader_options')" instead.
cm_redo	**Signature**: void **cm_redo** **Description**: 'cm_redo' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_redo')" instead.
cm_redraw	**Signature**: void **cm_redraw** **Description**: 'cm_redraw' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_redraw')" instead.

cm_reload	**Signature**: void **cm_reload** **Description**: 'cm_reload' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_reload')" instead.
cm_reset_window_state	**Signature**: void **cm_reset_window_state** **Description**: 'cm_reset_window_state' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_reset_window_state')" instead.
cm_restore_session	**Signature**: void **cm_restore_session** **Description**: 'cm_restore_session' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_restore_session')" instead.
cm_save	**Signature**: void **cm_save** **Description**: 'cm_save' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_save')" instead.
cm_save_all	**Signature**: void **cm_save_all** **Description**: 'cm_save_all' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_save_all')" instead.
cm_save_as	**Signature**: void **cm_save_as** **Description**: 'cm_save_as' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_save_as')" instead.
cm_save_bookmarks	**Signature**: void **cm_save_bookmarks** **Description**: 'cm_save_bookmarks' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_save_bookmarks')" instead.
cm_save_current_cell_as	**Signature**: void **cm_save_current_cell_as** **Description**: 'cm_save_current_cell_as' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_save_current_cell_as')" instead.
cm_save_layer_props	**Signature**: void **cm_save_layer_props** **Description**: 'cm_save_layer_props' action. Use of this method is deprecated

	This method is deprecated in version 0.27. Use "call_menu('cm_save_layer_props')" instead.
cm_save_session	**Signature**: void **cm_save_session** **Description**: 'cm_save_session' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_save_session')" instead.
cm_screenshot	**Signature**: void **cm_screenshot** **Description**: 'cm_screenshot' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_screenshot')" instead.
cm_screenshot_to_clipboard	**Signature**: void **cm_screenshot_to_clipboard** **Description**: 'cm_screenshot_to_clipboard' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_screenshot_to_clipboard')" instead.
cm_sel_flip_x	**Signature**: void **cm_sel_flip_x** **Description**: 'cm_sel_flip_x' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_sel_flip_x')" instead.
cm_sel_flip_y	**Signature**: void **cm_sel_flip_y** **Description**: 'cm_sel_flip_y' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_sel_flip_y')" instead.
cm_sel_free_rot	**Signature**: void **cm_sel_free_rot** **Description**: 'cm_sel_free_rot' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_sel_free_rot')" instead.
cm_sel_move	**Signature**: void **cm_sel_move** **Description**: 'cm_sel_move' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_sel_move')" instead.
cm_sel_move_to	**Signature**: void **cm_sel_move_to** **Description**: 'cm_sel_move_to' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_sel_move_to')" instead.

cm_sel_rot_ccw	**Signature**: void **cm_sel_rot_ccw** **Description**: 'cm_sel_rot_ccw' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_sel_rot_ccw')" instead.
cm_sel_rot_cw	**Signature**: void **cm_sel_rot_cw** **Description**: 'cm_sel_rot_cw' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_sel_rot_cw')" instead.
cm_sel_scale	**Signature**: void **cm_sel_scale** **Description**: 'cm_sel_scale' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_sel_scale')" instead.
cm_select_all	**Signature**: void **cm_select_all** **Description**: 'cm_select_all' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_select_all')" instead.
cm_select_cell	**Signature**: void **cm_select_cell** **Description**: 'cm_select_cell' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_select_cell')" instead.
cm_select_current_cell	**Signature**: void **cm_select_current_cell** **Description**: 'cm_select_current_cell' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_select_current_cell')" instead.
cm_setup	**Signature**: void **cm_setup** **Description**: 'cm_setup' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_setup')" instead.
cm_show_properties	**Signature**: void **cm_show_properties** **Description**: 'cm_show_properties' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_show_properties')" instead.
cm_technologies	**Signature**: void **cm_technologies** **Description**: 'cm_technologies' action. Use of this method is deprecated

	This method is deprecated in version 0.27. Use "call_menu('cm_technologies')" instead.
cm_undo	**Signature**: void **cm_undo** **Description**: 'cm_undo' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_undo')" instead.
cm_unselect_all	**Signature**: void **cm_unselect_all** **Description**: 'cm_unselect_all' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_unselect_all')" instead.
cm_view_log	**Signature**: void **cm_view_log** **Description**: 'cm_view_log' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_view_log')" instead.
cm_zoom_fit	**Signature**: void **cm_zoom_fit** **Description**: 'cm_zoom_fit' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_zoom_fit')" instead.
cm_zoom_fit_sel	**Signature**: void **cm_zoom_fit_sel** **Description**: 'cm_zoom_fit_sel' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_zoom_fit_sel')" instead.
cm_zoom_in	**Signature**: void **cm_zoom_in** **Description**: 'cm_zoom_in' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_zoom_in')" instead.
cm_zoom_out	**Signature**: void **cm_zoom_out** **Description**: 'cm_zoom_out' action. Use of this method is deprecated This method is deprecated in version 0.27. Use "call_menu('cm_zoom_out')" instead.
commit_config	**Signature**: void **commit_config** **Description**: Commits the configuration settings This method is provided for using MainWindow without an Application object. It's a convience method which is equivalent to 'dispatcher().commit_config(...)'. See Dispatcher#commit_config for details. This method has been introduced in version 0.27.

create_layout

(1) Signature: CellView **create_layout** (int mode)

Description: Creates a new, empty layout

mode: An integer value of 0, 1 or 2 that determines how the layout is created

Returns: The cellview of the layout that was created

Create the layout in the current view, replacing the current layouts (mode 0), in a new view (mode 1) or adding it to the current view (mode 2). In mode 1, the new view is made the current one.

This version uses the initial technology and associates it with the new layout.

Starting with version 0.25, this method returns a cellview object that can be modified to configure the cellview.

(2) Signature: CellView **create_layout** (string tech, int mode)

Description: Creates a new, empty layout with the given technology

mode: An integer value of 0, 1 or 2 that determines how the layout is created

tech: The name of the technology to use for that layout.

Returns: The cellview of the layout that was created

Create the layout in the current view, replacing the current layouts (mode 0), in a new view (mode 1) or adding it to the current view (mode 2). In mode 1, the new view is made the current one.

If the technology name is not a valid technology name, the default technology will be used.

This version was introduced in version 0.22. Starting with version 0.25, this method returns a cellview object that can be modified to configure the cellview.

create_view

Signature: int **create_view**

Description: Creates a new, empty view

Returns: The index of the view that was created

Creates an empty view that can be filled with layouts using the load_layout and create_layout methods on the view object. Use the view method to obtain the view object from the view index. This method has been added in version 0.22.

current_view

Signature: LayoutView ptr **current_view**

Description: Returns a reference to the current view's object

Returns: A reference to a LayoutView object representing the current view.

current_view_index

Signature: *[const]* int **current_view_index**

Description: Returns the current view's index

Returns: The index of the current view

This method will return the index of the current view.

Python specific notes:
The object exposes a readable attribute 'current_view_index'. This is the getter.

current_view_index=

Signature: void **current_view_index=** (int index)

Description: Selects the view with the given index

index: The index of the view to select (0 is the first)

This method will make the view with the given index the current (front) view.

This method was renamed from select_view to current_view_index= in version 0.25. The old name is still available, but deprecated.

Python specific notes:
The object exposes a writable attribute 'current_view_index'. This is the setter.

dispatcher

Signature: *[const]* Dispatcher ptr **dispatcher**

Description: Gets the dispatcher interface (the plugin root configuration space)

This method has been introduced in version 0.27.

enable_edits

Signature: void **enable_edits** (bool enable)

Description: Enables or disables editing

enable: Enable edits if set to true

Use of this method is deprecated

Starting from version 0.25, this method enables/disables edits on the current view only. Use LayoutView#enable_edits instead.

exit

Signature: void **exit**

Description: Schedules an exit for the application

This method does not immediately exit the application but sends an exit request to the application which will cause a clean shutdown of the GUI.

get_config

Signature: variant **get_config** (string name)

Description: Gets the value of a local configuration parameter

This method is provided for using MainWindow without an Application object. It's a convience method which is equivalent to 'dispatcher().get_config(...)'. See Dispatcher#get_config for details.

This method has been introduced in version 0.27.

get_config_names

Signature: string[] **get_config_names**

Description: Gets the configuration parameter names

This method is provided for using MainWindow without an Application object. It's a convience method which is equivalent to 'dispatcher().get_config_names(...)'. See Dispatcher#get_config_names for details.

This method has been introduced in version 0.27.

get_default_key_bindings

Signature: map<string,string> **get_default_key_bindings**

Description: Gets the default key bindings

This method returns a hash with the default key binding vs. menu item path. You can use this hash with set_key_bindings to reset all key bindings to the default ones.

This method has been introduced in version 0.27.

get_default_menu_items_hidden

Signature: map<string,bool> **get_default_menu_items_hidden**

Description: Gets the flags indicating whether menu items are hidden by default

You can use this hash with set_menu_items_hidden to restore the visibility of all menu items.

This method has been introduced in version 0.27.

get_key_bindings

Signature: map<string,string> **get_key_bindings**

Description: Gets the current key bindings

This method returns a hash with the key binding vs. menu item path.

This method has been introduced in version 0.27.

get_menu_items_hidden

Signature: map<string,bool> **get_menu_items_hidden**

Description: Gets the flags indicating whether menu items are hidden

This method returns a hash with the hidden flag vs. menu item path. You can use this hash with set_menu_items_hidden.

This method has been introduced in version 0.27.

grid_micron

Signature: *[const]* double **grid_micron**

Description: Gets the global grid in micron

Returns: The global grid in micron

The global grid is used at various places, i.e. for ruler snapping, for grid display etc.

index_of

Signature: *[const]* int **index_of** (const LayoutView ptr view)

Description: Gets the index of the given view

Returns: The index of the view that was given

If the given view is not a view object within the main window, a negative value will be returned.

This method has been added in version 0.25.

initial_technology

Signature: string **initial_technology**

Description: Gets the technology used for creating or loading layouts (unless explicitly specified)

Returns: The current initial technology

This method was added in version 0.22.

Python specific notes:
The object exposes a readable attribute 'initial_technology'. This is the getter.

initial_technology=

Signature: void **initial_technology=** (string tech)

Description: Sets the technology used for creating or loading layouts (unless explicitly specified)

tech: The new initial technology

Setting the technology will have an effect on the next load_layout or create_layout operation which does not explicitly specify the technology but might not be reflected correctly in the reader options dialog and changes will be reset when the application is restarted.

This method was added in version 0.22.

Python specific notes:

The object exposes a writable attribute 'initial_technology'. This is the setter.

instance

Signature: *[static]* MainWindow ptr **instance**

Description: Gets application's main window instance

This method has been added in version 0.24.

load_layout

(1) Signature: CellView **load_layout** (string filename, int mode = 1)

Description: Loads a new layout

filename:	The name of the file to load
mode:	An integer value of 0, 1 or 2 that determines how the file is loaded
Returns:	The cellview into which the layout was loaded

Loads the given file into the current view, replacing the current layouts (mode 0), into a new view (mode 1) or adding the layout to the current view (mode 2). In mode 1, the new view is made the current one.

This version will use the initial technology and the default reader options. Others versions are provided which allow specification of technology and reader options explicitly.

Starting with version 0.25, this method returns a cellview object that can be modified to configure the cellview. The 'mode' argument has been made optional in version 0.28.

(2) Signature: CellView **load_layout** (string filename, string tech, int mode = 1)

Description: Loads a new layout and associate it with the given technology

filename:	The name of the file to load
tech:	The name of the technology to use for that layout.
mode:	An integer value of 0, 1 or 2 that determines how the file is loaded
Returns:	The cellview into which the layout was loaded

Loads the given file into the current view, replacing the current layouts (mode 0), into a new view (mode 1) or adding the layout to the current view (mode 2). In mode 1, the new view is made the current one.

If the technology name is not a valid technology name, the default technology will be used. The 'mode' argument has been made optional in version 0.28.

This version was introduced in version 0.22. Starting with version 0.25, this method returns a cellview object that can be modified to configure the cellview.

(3) Signature: CellView **load_layout** (string filename, const LoadLayoutOptions options, int mode = 1)

Description: Loads a new layout with the given options

filename:	The name of the file to load
options:	The reader options to use.
mode:	An integer value of 0, 1 or 2 that determines how the file is loaded
Returns:	The cellview into which the layout was loaded

Loads the given file into the current view, replacing the current layouts (mode 0), into a new view (mode 1) or adding the layout to the current view (mode 2). In mode 1, the new view is made the current one.

This version was introduced in version 0.22. Starting with version 0.25, this method returns a cellview object that can be modified to configure the cellview. The 'mode' argument has been made optional in version 0.28.

(4) Signature: CellView **load_layout** (string filename, const LoadLayoutOptions options, string tech, int mode = 1)

Description: Loads a new layout with the given options and associate it with the given technology

filename:	The name of the file to load
options:	The reader options to use.
tech:	The name of the technology to use for that layout.
mode:	An integer value of 0, 1 or 2 that determines how the file is loaded
Returns:	The cellview into which the layout was loaded

Loads the given file into the current view, replacing the current layouts (mode 0), into a new view (mode 1) or adding the layout to the current view (mode 2). In mode 1, the new view is made the current one.

If the technology name is not a valid technology name, the default technology will be used.

This version was introduced in version 0.22. Starting with version 0.25, this method returns a cellview object that can be modified to configure the cellview. The 'mode' argument has been made optional in version 0.28.

manager

Signature: Manager **manager**

Description: Gets the Manager object of this window

The manager object is responsible to managing the undo/redo stack. Usually this object is not required. It's more convenient and safer to use the related methods provided by LayoutView (LayoutView#transaction, LayoutView#commit) and MainWindow (such as MainWindow#cm_undo and MainWindow#cm_redo).

This method has been added in version 0.24.

menu

Signature: AbstractMenu ptr **menu**

Description: Returns a reference to the abstract menu

Returns:	A reference to an AbstractMenu object representing the menu system

menu_symbols

Signature: *[static]* string[] **menu_symbols**

Description: Gets all available menu symbols (see call_menu).

NOTE: currently this method delivers a superset of all available symbols. Depending on the context, no all symbols may trigger actual functionality.

This method has been introduced in version 0.27.

message

Signature: void **message** (string message, int time = infinite)

Description: Displays a message in the status bar

message:	The message to display
time:	The time how long to display the message in ms. A negative value means 'infinitely'.

This given message is shown in the status bar for the given time.

This method has been added in version 0.18. The 'time' parameter was made optional in version 0.28.10.

on_current_view_changed

Signature: *[signal]* void **on_current_view_changed**

Description: An event indicating that the current view has changed

This event is triggered after the current view has changed. This happens, if the user switches the layout tab.

Before version 0.25 this event was based on the observer pattern obsolete now. The corresponding methods (add_current_view_observer/remove_current_view_observer) have been removed in 0.25.

Python specific notes:
The object exposes a readable attribute 'on_current_view_changed'. This is the getter.
The object exposes a writable attribute 'on_current_view_changed'. This is the setter.

on_session_about_to_be_restored

Signature: *[signal]* void **on_session_about_to_be_restored**

Description: An event indicating that a session is about to be restored

This event has been added in version 0.28.8.

Python specific notes:
The object exposes a readable attribute 'on_session_about_to_be_restored'. This is the getter.
The object exposes a writable attribute 'on_session_about_to_be_restored'. This is the setter.

on_session_restored

Signature: *[signal]* void **on_session_restored**

Description: An event indicating that a session was restored

This event has been added in version 0.28.8.

Python specific notes:
The object exposes a readable attribute 'on_session_restored'. This is the getter.
The object exposes a writable attribute 'on_session_restored'. This is the setter.

on_view_closed

Signature: *[signal]* void **on_view_closed** (int index)

Description: An event indicating that a view was closed

index: The index of the view that was closed

This event is triggered after a view was closed. For example, because the tab was closed.

This event has been added in version 0.25.

Python specific notes:
The object exposes a readable attribute 'on_view_closed'. This is the getter.
The object exposes a writable attribute 'on_view_closed'. This is the setter.

on_view_created

Signature: *[signal]* void **on_view_created** (int index)

Description: An event indicating that a new view was created

index: The index of the view that was created

This event is triggered after a new view was created. For example, if a layout is loaded into a new panel.

Before version 0.25 this event was based on the observer pattern obsolete now. The corresponding methods (add_new_view_observer/remove_new_view_observer) have been removed in 0.25.

Python specific notes:
The object exposes a readable attribute 'on_view_created'. This is the getter.

The object exposes a writable attribute 'on_view_created'. This is the setter.

read_config

Signature: bool **read_config** (string file_name)

Description: Reads the configuration from a file

This method is provided for using MainWindow without an Application object. It's a convience method which is equivalent to 'dispatcher().read_config(...)'. See Dispatcher#read_config for details.

This method has been introduced in version 0.27.

redraw

Signature: void **redraw**

Description: Redraws the current view

Issues a redraw request to the current view. This usually happens automatically, so this method does not need to be called in most relevant cases.

resize

Signature: void **resize** (int width, int height)

Description: Resizes the window

width: The new width of the window

height: The new width of the window

This method resizes the window to the given target size including decoration such as menu bar and control panels

restore_session

Signature: void **restore_session** (string fn)

Description: Restores a session from the given file

fn: The path to the session file

The session stored in the given session file is restored. All existing views are closed and all layout edits are discarded without notification.

This method was added in version 0.18.

save_session

Signature: void **save_session** (string fn)

Description: Saves the session to the given file

fn: The path to the session file

The session is saved to the given session file. Any existing layout edits are not automatically saved together with the session. The session just holds display settings and annotation objects. If layout edits exist, they have to be saved explicitly in a separate step.

This method was added in version 0.18.

select_view

Signature: void **select_view** (int index)

Description: Selects the view with the given index

index: The index of the view to select (0 is the first)

Use of this method is deprecated. Use current_view_index= instead

This method will make the view with the given index the current (front) view.

This method was renamed from select_view to current_view_index= in version 0.25. The old name is still available, but deprecated.

Python specific notes:

The object exposes a writable attribute 'current_view_index'. This is the setter.

set_config

Signature: void **set_config** (string name, string value)

Description: Set a local configuration parameter with the given name to the given value

This method is provided for using MainWindow without an Application object. It's a convience method which is equivalent to 'dispatcher().set_config(...)'. See Dispatcher#set_config for details.

This method has been introduced in version 0.27.

set_key_bindings

Signature: void **set_key_bindings** (map<string,string> bindings)

Description: Sets key bindings.

Sets the given key bindings. Pass a hash listing the key bindings per menu item paths. Key strings follow the usual notation, e.g. 'Ctrl+A', 'Shift+X' or just 'F2'. Use an empty value to remove a key binding from a menu entry.

get_key_bindings will give you the current key bindings, get_default_key_bindings will give you the default ones.

Examples:

```
# reset all key bindings to default:
mw = RBA::MainWindow.instance()
mw.set_key_bindings(mw.get_default_key_bindings())

# disable key binding for 'copy':
RBA::MainWindow.instance.set_key_bindings({ "edit_menu.copy" => "" })

# configure 'copy' to use Shift+K and 'cut' to use Ctrl+K:
RBA::MainWindow.instance.set_key_bindings({ "edit_menu.copy" => "Shift+K",
 "edit_menu.cut" => "Ctrl+K" })
```

This method has been introduced in version 0.27.

set_menu_items_hidden

Signature: void **set_menu_items_hidden** (map<string,bool> flags)

Description: sets the flags indicating whether menu items are hidden

This method allows hiding certain menu items. It takes a hash with hidden flags vs. menu item paths. Examples:

```
# show all menu items:
mw = RBA::MainWindow.instance()
mw.set_menu_items_hidden(mw.get_default_menu_items_hidden())

# hide the 'copy' entry from the 'Edit' menu:
RBA::MainWindow.instance().set_menu_items_hidden({ "edit_menu.copy" =>
 true })
```

This method has been introduced in version 0.27.

show_macro_editor

Signature: void **show_macro_editor** (string cat = , bool add = false)

Description: Shows the macro editor

If 'cat' is given, this category will be selected in the category tab. If 'add' is true, the 'new macro' dialog will be opened.

This method has been introduced in version 0.26.

synchronous

(1) Signature: void **synchronous** (bool sync_mode)

Description: Puts the main window into synchronous mode

sync_mode: 'true' if the application should behave synchronously

Use of this method is deprecated. Use synchronous= instead

In synchronous mode, an application is allowed to block on redraw. While redrawing, no user interactions are possible. Although this is not desirable for smooth operation, it can be beneficial for test or automation purposes, i.e. if a screenshot needs to be produced once the application has finished drawing.

Python specific notes:
The object exposes a writable attribute 'synchronous'. This is the setter.
This method is available as 'synchronous_' in Python to distiguish it from the property with the same name.

(2) Signature: *[const]* bool **synchronous**

Description: Gets a value indicating whether synchronous mode is activated

See synchronous= for details about this attribute

This property getter was introduced in version 0.29.

Python specific notes:
The object exposes a readable attribute 'synchronous'. This is the getter.

synchronous=

Signature: void **synchronous=** (bool sync_mode)

Description: Puts the main window into synchronous mode

sync_mode: 'true' if the application should behave synchronously

In synchronous mode, an application is allowed to block on redraw. While redrawing, no user interactions are possible. Although this is not desirable for smooth operation, it can be beneficial for test or automation purposes, i.e. if a screenshot needs to be produced once the application has finished drawing.

Python specific notes:
The object exposes a writable attribute 'synchronous'. This is the setter.
This method is available as 'synchronous_' in Python to distiguish it from the property with the same name.

title

Signature: *[const]* string **title**

Description: Gets the window title

See title= for a description of this property. This property was introduced in version 0.29.

Python specific notes:
The object exposes a readable attribute 'title'. This is the getter.

title=

Signature: void **title=** (string title)

Description: Sets the window title

If the window title is not empty, it will be used for the application window's title. Otherwise the default title is used. The title string is subject to expression interpolation. So it is possible to implement the default scheme of adding the current view using the following code:

```
add_view_info = "$(var view=LayoutView.current; view ? ' - ' +
 (view.is_dirty ? '[+] ' : '') + view.title : '')"
```

```
RBA::MainWindow.instance.title = "Custom Title" + add_view_info
```

This property was introduced in version 0.29.

Python specific notes:
The object exposes a writable attribute 'title'. This is the setter.

view

Signature: LayoutView ptr **view** (int n)

Description: Returns a reference to a view object by index

Returns: The view object's reference for the view with the given index.

views

Signature: *[const]* unsigned int **views**

Description: Returns the number of views

Returns: The number of views available so far.

write_config

Signature: bool **write_config** (string file_name)

Description: Writes configuration to a file

This method is provided for using MainWindow without an Application object. It's a convience method which is equivalent to 'dispatcher().write_config(...)'. See Dispatcher#write_config for details.

This method has been introduced in version 0.27.

4.233. API reference - Class Application

Notation used in Ruby API documentation

Module: lay

Description: The application object

Class hierarchy: Application » QCoreApplication » QObject

The application object is the main port from which to access all the internals of the application, in particular the main window.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add_macro_category**	(string name, string description, string[] folders)	Creates a new macro category
[const]	string	**application_data_path**		Returns the application's data path (where the configuration file is stored for example)
	string	**arch**		Returns the architecture string
	void	**commit_config**		Commits the configuration settings
	int	**execute**		Executes the application's main loop
	void	**exit**	(int result)	Ends the application with the given exit status
[const]	string	**get_config**	(string name)	Gets the value for a configuration parameter
[const]	string[]	**get_config_names**		Gets the configuration parameter names
[const]	string	**inst_path**		Returns the application's installation path (where the executable is located)
[const]	bool	**is_editable?**		Returns true if the application is in editable mode
[const]	string[]	**klayout_path**		Returns the KLayout path (search path for KLayout components)

[const]	MainWindow ptr	**main_window**		Returns a reference to the main window
[signal]	void	**on_salt_changed**		This event is triggered when the package status changes.
	void	**process_events**		Processes pending events
	bool	**read_config**	(string file_name)	Reads the configuration from a file
	void	**set_config**	(string name, string value)	Sets a configuration parameter with the given name to the given value
	string	**version**		Returns the application's version string
	bool	**write_config**	(string file_name)	Writes configuration to a file

Public static methods and constants

Application ptr	**instance**	Return the singleton instance of the application

Deprecated methods (protected, public, static, non-static and constructors)

int	**exec**	Use of this method is deprecated. Use execute instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add_macro_category

Signature: void **add_macro_category** (string name, string description, string[] folders)

Description: Creates a new macro category

Creating a new macro category is only possible during the autorun_early stage. The new macro category must correspond to an interpreter registered at the same stage. This method has been introduced in version 0.28.

application_data_path

Signature: *[const]* string **application_data_path**

Description: Returns the application's data path (where the configuration file is stored for example)

This method has been added in version 0.22.

arch

Signature: string **arch**

Description: Returns the architecture string

This method has been introduced in version 0.25.

commit_config

Signature: void **commit_config**

Description: Commits the configuration settings

Some configuration options are queued for performance reasons and become active only after 'commit_config' has been called. After a sequence of set_config calls, this method should be called to activate the settings made by these calls.

This method has been introduced in version 0.25.

exec

Signature: int **exec**

Description: Executes the application's main loop

Use of this method is deprecated. Use execute instead

This method must be called in order to execute the application in the main script if a script is provided.

Python specific notes:
This attribute is available as 'exec_' in Python.

execute

Signature: int **execute**

Description: Executes the application's main loop

This method must be called in order to execute the application in the main script if a script is provided.

Python specific notes:
This attribute is available as 'exec_' in Python.

exit

Signature: void **exit** (int result)

Description: Ends the application with the given exit status

This method should be called instead of simply shutting down the process. It performs some important cleanup without which the process might crash. If the result code is 0 (success), the configuration file will be updated unless that has been disabled by the -nc command line switch. This method has been added in version 0.22.

get_config

Signature: *[const]* string **get_config** (string name)

Description: Gets the value for a configuration parameter

name:	The name of the configuration parameter whose value shall be obtained (a string)
Returns:	The value of the parameter

This method returns the value of the given configuration parameter. If the parameter is not known, an exception will be thrown. Use get_config_names to obtain a list of all configuration parameter names available.

Configuration parameters are always stored as strings. The actual format of this string is specific to the configuration parameter. The values delivered by this method correspond to the values stored in the configuration file

get_config_names

Signature: *[const]* string[] **get_config_names**

Description: Gets the configuration parameter names

Returns:	A list of configuration parameter names

This method returns the names of all known configuration parameters. These names can be used to get and set configuration parameter values.

inst_path

Signature: *[const]* string **inst_path**

Description: Returns the application's installation path (where the executable is located)

This method has been added in version 0.18. Version 0.22 offers the method klayout_path which delivers all components of the search path.

instance

Signature: *[static]* Application ptr **instance**

Description: Return the singleton instance of the application

There is exactly one instance of the application. This instance can be obtained with this method.

is_editable?

Signature: *[const]* bool **is_editable?**

Description: Returns true if the application is in editable mode

klayout_path

Signature: *[const]* string[] **klayout_path**

Description: Returns the KLayout path (search path for KLayout components)

The result is an array containing the components of the path.

This method has been added in version 0.22.

main_window

Signature: *[const]* MainWindow ptr **main_window**

Description: Returns a reference to the main window

Returns: A object reference to the main window object.

on_salt_changed

Signature: *[signal]* void **on_salt_changed**

Description: This event is triggered when the package status changes.

Register to this event if you are interested in package changes - i.e. installation or removal of packages or package updates.

This event has been introduced in version 0.28.

Python specific notes:
The object exposes a readable attribute 'on_salt_changed'. This is the getter.
The object exposes a writable attribute 'on_salt_changed'. This is the setter.

process_events

Signature: void **process_events**

Description: Processes pending events

This method processes pending events and dispatches them internally. Calling this method periodically during a long operation keeps the application 'alive'

read_config

Signature: bool **read_config** (string file_name)

Description: Reads the configuration from a file

Returns: A value indicating whether the operation was successful

This method silently does nothing, if the config file does not exist. If it does and an error occurred, the error message is printed on stderr. In both cases, false is returned.

set_config

Signature: void **set_config** (string name, string value)

Description: Sets a configuration parameter with the given name to the given value

name: The name of the configuration parameter to set

value: The value to which to set the configuration parameter

This method sets the configuration parameter with the given name to the given value. Values can only be strings. Numerical values have to be converted into strings first. The actual format of the value depends on the configuration parameter. The name must be one of the names returned by get_config_names. It is possible to write an arbitrary name/value pair into the configuration database which then is written to the configuration file.

version

Signature: string **version**

Description: Returns the application's version string

write_config

Signature: bool **write_config** (string file_name)

Description: Writes configuration to a file

Returns: A value indicating whether the operation was successful

If the configuration file cannot be written, is returned but no exception is thrown.

4.234. API reference - Class LEFDEFReaderConfiguration

Notation used in Ruby API documentation

Module: db

Description: Detailed LEF/DEF reader options

This class is a aggregate belonging to the LoadLayoutOptions class. It provides options for the LEF/DEF reader. These options have been placed into a separate class to account for their complexity. This class specifically handles layer mapping. This is the process of generating layer names or GDS layer/datatypes from LEF/DEF layers and purpose combinations. There are basically two ways: to use a map file or to use pattern-based production rules.

To use a layer map file, set the map_file attribute to the name of the layer map file. The layer map file lists the GDS layer and datatype numbers to generate for the geometry.

The pattern-based approach will use the layer name and attach a purpose-dependent suffix to it. Use the ..._suffix attributes to specify this suffix. For routing, the corresponding attribute is routing_suffix for example. A purpose can also be mapped to a specific GDS datatype using the corresponding ..._datatype attributes. The decorated or undecorated names are looked up in a layer mapping table in the next step. The layer mapping table is specified using the layer_map attribute. This table can be used to map layer names to specific GDS layers by using entries of the form 'NAME: layer-number'.

If a layer map file is present, the pattern-based attributes are ignored.

Public constructors

new LEFDEFReaderConfiguration ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const LEFDEFReaderC other)	Assigns another object to self
[const]	int	**blockages_datatype**		Gets the blockage marker layer datatype value.
	void	**blockages_datatype=**	(int datatype)	Sets the blockage marker layer datatype value.
[const]	string	**blockages_suffix**		Gets the blockage marker layer name suffix.
	void	**blockages_suffix=**	(string suffix)	Sets the blockage marker layer name suffix.

[const]	string	**cell_outline_layer**		Gets the layer on which to produce the cell outline (diearea).
	void	**cell_outline_layer=**	(string spec)	Sets the layer on which to produce the cell outline (diearea).
	void	**clear_fill_datatypes_per_mask**		Clears the fill layer datatypes per mask.
	void	**clear_fills_suffixes_per_mas**		Clears the fill layer name suffix per mask.
	void	**clear_lef_pins_datatypes_per_mask**		Clears the LEF pin layer datatypes per mask.
	void	**clear_lef_pins_suffixes_per_**		Clears the LEF pin layer name suffix per mask.
	void	**clear_pin_datatypes_per_mask**		Clears the pin layer datatypes per mask.
	void	**clear_pins_suffixes_per_ma**		Clears the pin layer name suffix per mask.
	void	**clear_routing_datatypes_per_mask**		Clears the routing layer datatypes per mask.
	void	**clear_routing_suffixes_per_**		Clears the routing layer name suffix per mask.
	void	**clear_special_routing_datatypes_per_mask**		Clears the special routing layer datatypes per mask.
	void	**clear_special_routing_suffix**		Clears the special routing layer name suffix per mask.
	void	**clear_via_geometry_datatypes_per_mask**		Clears the via geometry layer datatypes per mask.
	void	**clear_via_geometry_suffixes**		Clears the via geometry layer name suffix per mask.
[const]	bool	**create_other_layers**		Gets a value indicating whether layers not mapped in the layer map shall be created too
	void	**create_other_layers=**	(bool f)	Sets a value indicating whether layers not mapped in the layer map shall be created too
[const]	double	**dbu**		Gets the database unit to use for producing the layout.
	void	**dbu=**	(double dbu)	Sets the database unit to use for producing the layout.
[const]	new LEFDEFReaderConfiguration ptr	**dup**		Creates a copy of self
[const]	int	**fills_datatype**		Gets the fill geometry layer datatype value.
[const]	int	**fills_datatype**	(unsigned int mask)	Gets the fill geometry layer datatype value per mask.
	void	**fills_datatype=**	(int datatype)	Sets the fill geometry layer datatype value.
[const]	string	**fills_suffix**		Gets the fill geometry layer name suffix.

	void	fills_suffix=	(string suffix)	Sets the fill geometry layer name suffix.
[const]	string	fills_suffix_per_mask	(unsigned int mask)	Gets the fill geometry layer name suffix per mask.
[const]	variant	instance_property_name		Gets a value indicating whether and how to produce instance names as properties.
	void	instance_property_name=	(variant name)	Sets a value indicating whether and how to produce instance names as properties.
[const]	bool	joined_paths		Gets a value indicating whether to create joined paths for wires.
	void	joined_paths=	(bool flag)	Sets a value indicating whether to create joined paths for wires.
[const]	int	labels_datatype		Gets the labels layer datatype value.
	void	labels_datatype=	(int datatype)	Sets the labels layer datatype value.
[const]	string	labels_suffix		Gets the label layer name suffix.
	void	labels_suffix=	(string suffix)	Sets the label layer name suffix.
	LayerMap	layer_map		Gets the layer map to be used for the LEF/DEF reader
	void	layer_map=	(const LayerMap m)	Sets the layer map to be used for the LEF/DEF reader
[const]	string[]	lef_files		Gets the list technology LEF files to additionally import
	void	lef_files=	(string[] lef_file_paths)	Sets the list technology LEF files to additionally import
[const]	int	lef_labels_datatype		Gets the lef_labels layer datatype value.
	void	lef_labels_datatype=	(int datatype)	Sets the lef_labels layer datatype value.
[const]	string	lef_labels_suffix		Gets the label layer name suffix.
	void	lef_labels_suffix=	(string suffix)	Sets the label layer name suffix.
[const]	int	lef_pins_datatype		Gets the LEF pin geometry layer datatype value.
[const]	int	lef_pins_datatype	(unsigned int mask)	Gets the LEF pin geometry layer datatype value per mask.
	void	lef_pins_datatype=	(int datatype)	Sets the LEF pin geometry layer datatype value.
[const]	string	lef_pins_suffix		Gets the LEF pin geometry layer name suffix.
	void	lef_pins_suffix=	(string suffix)	Sets the LEF pin geometry layer name suffix.

[const]	string	**lef_pins_suffix_per_mask**	(unsigned int mask)	Gets the LEF pin geometry layer name suffix per mask.
[const]	string[]	**macro_layout_files**		Gets the list of layout files to read for substituting macros in DEF
	void	**macro_layout_files=**	(string[] file_paths)	Sets the list of layout files to read for substituting macros in DEF
[const]	Layout ptr[]	**macro_layouts**		Gets the layout objects used for resolving LEF macros in the DEF reader.
	void	**macro_layouts=**	(Layout ptr[] layouts)	Sets the layout objects used for resolving LEF macros in the DEF reader.
[const]	unsigned int	**macro_resolution_mode**		Gets the macro resolution mode (LEF macros into DEF).
	void	**macro_resolution_mode=**	(unsigned int mode)	Sets the macro resolution mode (LEF macros into DEF).
[const]	string	**map_file**		Gets the layer map file to use.
	void	**map_file=**	(string file)	Sets the layer map file to use.
[const]	variant	**net_property_name**		Gets a value indicating whether and how to produce net names as properties.
	void	**net_property_name=**	(variant name)	Sets a value indicating whether and how to produce net names as properties.
[const]	int	**obstructions_datatype**		Gets the obstruction marker layer datatype value.
	void	**obstructions_datatype=**	(int datatype)	Sets the obstruction marker layer datatype value.
[const]	string	**obstructions_suffix**		Gets the obstruction marker layer name suffix.
	void	**obstructions_suffix=**	(string suffix)	Sets the obstruction marker layer name suffix.
	void	**paths_relative_to_cwd=**	(bool f)	Sets a value indicating whether to use paths relative to cwd (true) or DEF file (false) for map or LEF files
[const]	variant	**pin_property_name**		Gets a value indicating whether and how to produce pin names as properties.
	void	**pin_property_name=**	(variant name)	Sets a value indicating whether and how to produce pin names as properties.
[const]	int	**pins_datatype**		Gets the pin geometry layer datatype value.
[const]	int	**pins_datatype**	(unsigned int mask)	Gets the pin geometry layer datatype value per mask.
	void	**pins_datatype=**	(int datatype)	Sets the pin geometry layer datatype value.

[const]	string	**pins_suffix**		Gets the pin geometry layer name suffix.
	void	**pins_suffix=**	(string suffix)	Sets the pin geometry layer name suffix.
[const]	string	**pins_suffix_per_mask**	(unsigned int mask)	Gets the pin geometry layer name suffix per mask.
[const]	string	**placement_blockage_layer**		Gets the layer on which to produce the placement blockage.
	void	**placement_blockage_layer=**	(string layer)	Sets the layer on which to produce the placement blockage.
[const]	bool	**produce_blockages**		Gets a value indicating whether routing blockage markers shall be produced.
	void	**produce_blockages=**	(bool produce)	Sets a value indicating whether routing blockage markers shall be produced.
[const]	bool	**produce_cell_outlines**		Gets a value indicating whether to produce cell outlines (diearea).
	void	**produce_cell_outlines=**	(bool produce)	Sets a value indicating whether to produce cell outlines (diearea).
[const]	bool	**produce_fills**		Gets a value indicating whether fill geometries shall be produced.
	void	**produce_fills=**	(bool produce)	Sets a value indicating whether fill geometries shall be produced.
[const]	bool	**produce_labels**		Gets a value indicating whether labels shall be produced.
	void	**produce_labels=**	(bool produce)	Sets a value indicating whether labels shall be produced.
[const]	bool	**produce_lef_labels**		Gets a value indicating whether lef_labels shall be produced.
	void	**produce_lef_labels=**	(bool produce)	Sets a value indicating whether lef_labels shall be produced.
[const]	bool	**produce_lef_pins**		Gets a value indicating whether LEF pin geometries shall be produced.
	void	**produce_lef_pins=**	(bool produce)	Sets a value indicating whether LEF pin geometries shall be produced.
[const]	bool	**produce_obstructions**		Gets a value indicating whether obstruction markers shall be produced.
	void	**produce_obstructions=**	(bool produce)	Sets a value indicating whether obstruction markers shall be produced.
[const]	bool	**produce_pins**		Gets a value indicating whether pin geometries shall be produced.

	void	produce_pins=	(bool produce)	Sets a value indicating whether pin geometries shall be produced.
[const]	bool	produce_placement_blockages		Gets a value indicating whether to produce placement blockage regions.
	void	produce_placement_blockag	(bool produce)	Sets a value indicating whether to produce placement blockage regions.
[const]	bool	produce_regions		Gets a value indicating whether to produce regions.
	void	produce_regions=	(bool produce)	Sets a value indicating whether to produce regions.
[const]	bool	produce_routing		Gets a value indicating whether routing geometry shall be produced.
	void	produce_routing=	(bool produce)	Sets a value indicating whether routing geometry shall be produced.
[const]	bool	produce_special_routing		Gets a value indicating whether special routing geometry shall be produced.
	void	produce_special_routing=	(bool produce)	Sets a value indicating whether special routing geometry shall be produced.
[const]	bool	produce_via_geometry		Sets a value indicating whether via geometries shall be produced.
	void	produce_via_geometry=	(bool produce)	Sets a value indicating whether via geometries shall be produced.
[const]	bool	read_lef_with_def		Gets a value indicating whether to read all LEF files in the same directory than the DEF file.
	void	read_lef_with_def=	(bool flag)	Sets a value indicating whether to read all LEF files in the same directory than the DEF file.
[const]	string	region_layer		Gets the layer on which to produce the regions.
	void	region_layer=	(string layer)	Sets the layer on which to produce the regions.
[const]	int	routing_datatype		Gets the routing layer datatype value.
[const]	int	routing_datatype	(unsigned int mask)	Gets the routing geometry layer datatype value per mask.
	void	routing_datatype=	(int datatype)	Sets the routing layer datatype value.
[const]	string	routing_suffix		Gets the routing layer name suffix.
	void	routing_suffix=	(string suffix)	Sets the routing layer name suffix.
[const]	string	routing_suffix_per_mask	(unsigned int mask)	Gets the routing geometry layer name suffix per mask.

[const]	bool	separate_groups		Gets a value indicating whether to create separate parent cells for individual groups.
	void	separate_groups=	(bool flag)	Sets a value indicating whether to create separate parent cells for individual groups.
	void	set_fills_datatype_per_mask	(unsigned int mask, int datatype)	Sets the fill geometry layer datatype value.
	void	set_fills_suffix_per_mask	(unsigned int mask, string suffix)	Sets the fill geometry layer name suffix per mask.
	void	set_lef_pins_datatype_per_mask	(unsigned int mask, int datatype)	Sets the LEF pin geometry layer datatype value.
	void	set_lef_pins_suffix_per_mas	(unsigned int mask, string suffix)	Sets the LEF pin geometry layer name suffix per mask.
	void	set_pins_datatype_per_mask	(unsigned int mask, int datatype)	Sets the pin geometry layer datatype value.
	void	set_pins_suffix_per_mask	(unsigned int mask, string suffix)	Sets the pin geometry layer name suffix per mask.
	void	set_routing_datatype_per_mask	(unsigned int mask, int datatype)	Sets the routing geometry layer datatype value.
	void	set_routing_suffix_per_masl	(unsigned int mask, string suffix)	Sets the routing geometry layer name suffix per mask.
	void	set_special_routing_datatype_per_mask	(unsigned int mask, int datatype)	Sets the special routing geometry layer datatype value.
	void	set_special_routing_suffix_p	(unsigned int mask, string suffix)	Sets the special routing geometry layer name suffix per mask.
	void	set_via_geometry_datatype_per_mask	(unsigned int mask, int datatype)	Sets the via geometry layer datatype value.
	void	set_via_geometry_suffix_pe	(unsigned int mask, string suffix)	Sets the via geometry layer name suffix per mask.
[const]	int	special_routing_datatype		Gets the special routing layer datatype value.
[const]	int	special_routing_datatype	(unsigned int mask)	Gets the special routing geometry layer datatype value per mask.

	void	special_routing_datatype=	(int datatype)	Sets the special routing layer datatype value.
[const]	string	special_routing_suffix		Gets the special routing layer name suffix.
	void	special_routing_suffix=	(string suffix)	Sets the special routing layer name suffix.
[const]	string	special_routing_suffix_per_m	(unsigned int mask)	Gets the special routing geometry layer name suffix per mask.
[const]	string	via_cellname_prefix		Gets the via cellname prefix.
	void	via_cellname_prefix=	(string prefix)	Sets the via cellname prefix.
[const]	int	via_geometry_datatype		Gets the via geometry layer datatype value.
[const]	int	via_geometry_datatype	(unsigned int mask)	Gets the via geometry layer datatype value per mask.
	void	via_geometry_datatype=	(int datatype)	Sets the via geometry layer datatype value.
[const]	string	via_geometry_suffix		Gets the via geometry layer name suffix.
	void	via_geometry_suffix=	(string suffix)	Sets the via geometry layer name suffix.
[const]	string	via_geometry_suffix_per_ma	(unsigned int mask)	Gets the via geometry layer name suffix per mask.

Deprecated methods (protected, public, static, non-static and constructors)

	void	create	Use of this method is deprecated. Use _create instead
	void	destroy	Use of this method is deprecated. Use _destroy instead
[const]	bool	destroyed?	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	is_const_object?	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const LEFDEFReaderConfiguration other)

Description: Assigns another object to self

blockages_datatype

Signature: *[const]* int **blockages_datatype**

Description: Gets the blockage marker layer datatype value.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'blockages_datatype'. This is the getter.

blockages_datatype=

Signature: void **blockages_datatype=** (int datatype)

Description: Sets the blockage marker layer datatype value.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'blockages_datatype'. This is the setter.

blockages_suffix

Signature: *[const]* string **blockages_suffix**

Description: Gets the blockage marker layer name suffix.

See produce_via_geometry for details about the layer production rules.

Python specific notes:

	The object exposes a readable attribute 'blockages_suffix'. This is the getter.
blockages_suffix=	**Signature**: void **blockages_suffix=** (string suffix) **Description**: Sets the blockage marker layer name suffix. See produce_via_geometry for details about the layer production rules. **Python specific notes:** The object exposes a writable attribute 'blockages_suffix'. This is the setter.
cell_outline_layer	**Signature**: *[const]* string **cell_outline_layer** **Description**: Gets the layer on which to produce the cell outline (diearea). This attribute is a string corresponding to the string representation of LayerInfo. This string can be either a layer number, a layer/datatype pair, a name or a combination of both. See LayerInfo for details. The setter for this attribute is cell_outline_layer=. See also produce_cell_outlines. **Python specific notes:** The object exposes a readable attribute 'cell_outline_layer'. This is the getter.
cell_outline_layer=	**Signature**: void **cell_outline_layer=** (string spec) **Description**: Sets the layer on which to produce the cell outline (diearea). See cell_outline_layer for details. **Python specific notes:** The object exposes a writable attribute 'cell_outline_layer'. This is the setter.
clear_fill_datatypes_per_mask	**Signature**: void **clear_fill_datatypes_per_mask** **Description**: Clears the fill layer datatypes per mask. See produce_via_geometry for details about this property. Mask specific rules have been introduced in version 0.27.
clear_fills_suffixes_per_mask	**Signature**: void **clear_fills_suffixes_per_mask** **Description**: Clears the fill layer name suffix per mask. See produce_via_geometry for details about this property. Mask specific rules have been introduced in version 0.27.
clear_lef_pins_datatypes_per_mask	**Signature**: void **clear_lef_pins_datatypes_per_mask** **Description**: Clears the LEF pin layer datatypes per mask. See produce_via_geometry for details about this property. Mask specific rules have been introduced in version 0.27.
clear_lef_pins_suffixes_per_ma	**Signature**: void **clear_lef_pins_suffixes_per_mask** **Description**: Clears the LEF pin layer name suffix per mask. See produce_via_geometry for details about this property. Mask specific rules have been introduced in version 0.27.
clear_pin_datatypes_per_mask	**Signature**: void **clear_pin_datatypes_per_mask** **Description**: Clears the pin layer datatypes per mask. See produce_via_geometry for details about this property.

	Mask specific rules have been introduced in version 0.27.
clear_pins_suffixes_per_mask	**Signature**: void **clear_pins_suffixes_per_mask** **Description**: Clears the pin layer name suffix per mask. See produce_via_geometry for details about this property. Mask specific rules have been introduced in version 0.27.
clear_routing_datatypes_per_mask	**Signature**: void **clear_routing_datatypes_per_mask** **Description**: Clears the routing layer datatypes per mask. See produce_via_geometry for details about this property. Mask specific rules have been introduced in version 0.27.
clear_routing_suffixes_per_ma	**Signature**: void **clear_routing_suffixes_per_mask** **Description**: Clears the routing layer name suffix per mask. See produce_via_geometry for details about this property. Mask specific rules have been introduced in version 0.27.
clear_special_routing_datatypes_per_mask	**Signature**: void **clear_special_routing_datatypes_per_mask** **Description**: Clears the special routing layer datatypes per mask. See produce_via_geometry for details about this property. Mask specific rules have been introduced in version 0.27.
clear_special_routing_suffixes_	**Signature**: void **clear_special_routing_suffixes_per_mask** **Description**: Clears the special routing layer name suffix per mask. See produce_via_geometry for details about this property. Mask specific rules have been introduced in version 0.27.
clear_via_geometry_datatypes_per_mask	**Signature**: void **clear_via_geometry_datatypes_per_mask** **Description**: Clears the via geometry layer datatypes per mask. See produce_via_geometry for details about this property. Mask specific rules have been introduced in version 0.27.
clear_via_geometry_suffixes_p	**Signature**: void **clear_via_geometry_suffixes_per_mask** **Description**: Clears the via geometry layer name suffix per mask. See produce_via_geometry for details about this property. Mask specific rules have been introduced in version 0.27.
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

create_other_layers

Signature: *[const]* bool **create_other_layers**

Description: Gets a value indicating whether layers not mapped in the layer map shall be created too

See layer_map for details.

Python specific notes:
The object exposes a readable attribute 'create_other_layers'. This is the getter.

create_other_layers=

Signature: void **create_other_layers=** (bool f)

Description: Sets a value indicating whether layers not mapped in the layer map shall be created too

See layer_map for details.

Python specific notes:
The object exposes a writable attribute 'create_other_layers'. This is the setter.

dbu

Signature: *[const]* double **dbu**

Description: Gets the database unit to use for producing the layout.

This value specifies the database to be used for the layout that is read. When a DEF file is specified with a different database unit, the layout is translated into this database unit.

Python specific notes:
The object exposes a readable attribute 'dbu'. This is the getter.

dbu=

Signature: void **dbu=** (double dbu)

Description: Sets the database unit to use for producing the layout.

See dbu for details.

Python specific notes:
The object exposes a writable attribute 'dbu'. This is the setter.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new LEFDEFReaderConfiguration ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

fills_datatype

(1) Signature: *[const]* int **fills_datatype**

Description: Gets the fill geometry layer datatype value.

See produce_via_geometry for details about the layer production rules.

Fill support has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'fills_datatype'. This is the getter.

(2) Signature: *[const]* int **fills_datatype** (unsigned int mask)

Description: Gets the fill geometry layer datatype value per mask.

See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...).

Mask specific rules have been introduced in version 0.27.

Python specific notes:
This method is available as 'fills_datatype_' in Python to distiguish it from the property with the same name.

fills_datatype=

Signature: void **fills_datatype=** (int datatype)

Description: Sets the fill geometry layer datatype value.

See produce_via_geometry for details about the layer production rules.

Fill support has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'fills_datatype'. This is the setter.

fills_suffix

Signature: *[const]* string **fills_suffix**

Description: Gets the fill geometry layer name suffix.

See produce_via_geometry for details about the layer production rules.

Fill support has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'fills_suffix'. This is the getter.

fills_suffix=

Signature: void **fills_suffix=** (string suffix)

Description: Sets the fill geometry layer name suffix.

See produce_via_geometry for details about the layer production rules.

Fill support has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'fills_suffix'. This is the setter.

fills_suffix_per_mask

Signature: *[const]* string **fills_suffix_per_mask** (unsigned int mask)

Description: Gets the fill geometry layer name suffix per mask.

See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...).

Mask specific rules have been introduced in version 0.27.

instance_property_name

Signature: *[const]* variant **instance_property_name**

Description: Gets a value indicating whether and how to produce instance names as properties.

	If set to a value not nil, instance names will be attached to the instances generated as user properties. This attribute then specifies the user property name to be used for attaching the instance names. If set to nil, no instance names will be produced. The corresponding setter is instance_property_name=. This method has been introduced in version 0.26.4. **Python specific notes:** The object exposes a readable attribute 'instance_property_name'. This is the getter.
instance_property_name=	**Signature**: void **instance_property_name=** (variant name) **Description**: Sets a value indicating whether and how to produce instance names as properties. See instance_property_name for details. This method has been introduced in version 0.26.4. **Python specific notes:** The object exposes a writable attribute 'instance_property_name'. This is the setter.
is_const_object?	**Signature**: *[const]* bool **is_const_object?** **Description**: Returns a value indicating whether the reference is a const reference Use of this method is deprecated. Use _is_const_object? instead This method returns true, if self is a const reference. In that case, only const methods may be called on self.
joined_paths	**Signature**: *[const]* bool **joined_paths** **Description**: Gets a value indicating whether to create joined paths for wires. If this property is set to true, wires are represented by multi-segment paths as far as possible (this will fail for 45 degree path segments for example). By defauls, wires are represented by multiple straight segments. This property has been added in version 0.28.8. **Python specific notes:** The object exposes a readable attribute 'joined_paths'. This is the getter.
joined_paths=	**Signature**: void **joined_paths=** (bool flag) **Description**: Sets a value indicating whether to create joined paths for wires. See joined_paths for details about this property. This property has been added in version 0.28.8. **Python specific notes:** The object exposes a writable attribute 'joined_paths'. This is the setter.
labels_datatype	**Signature**: *[const]* int **labels_datatype** **Description**: Gets the labels layer datatype value. See produce_via_geometry for details about the layer production rules. **Python specific notes:** The object exposes a readable attribute 'labels_datatype'. This is the getter.
labels_datatype=	**Signature**: void **labels_datatype=** (int datatype) **Description**: Sets the labels layer datatype value. See produce_via_geometry for details about the layer production rules. **Python specific notes:**

The object exposes a writable attribute 'labels_datatype'. This is the setter.

labels_suffix

Signature: *[const]* string **labels_suffix**

Description: Gets the label layer name suffix.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'labels_suffix'. This is the getter.

labels_suffix=

Signature: void **labels_suffix=** (string suffix)

Description: Sets the label layer name suffix.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'labels_suffix'. This is the setter.

layer_map

Signature: LayerMap **layer_map**

Description: Gets the layer map to be used for the LEF/DEF reader

Returns: A reference to the layer map

Because LEF/DEF layer mapping is substantially different than for normal layout files, the LEF/DEF reader employs a separate layer mapping table. The LEF/DEF specific layer mapping is stored within the LEF/DEF reader's configuration and can be accessed with this attribute. The layer mapping table of LoadLayoutOptions will be ignored for the LEF/DEF reader.

The setter is layer_map=. create_other_layers= is available to control whether layers not specified in the layer mapping table shall be created automatically.

Python specific notes:
The object exposes a readable attribute 'layer_map'. This is the getter.

layer_map=

Signature: void **layer_map=** (const LayerMap m)

Description: Sets the layer map to be used for the LEF/DEF reader

See layer_map for details.

Python specific notes:
The object exposes a writable attribute 'layer_map'. This is the setter.

lef_files

Signature: *[const]* string[] **lef_files**

Description: Gets the list technology LEF files to additionally import

Returns a list of path names for technology LEF files to read in addition to the primary file. Relative paths are resolved relative to the file to read or relative to the technology base path.

The setter for this property is lef_files=.

Python specific notes:
The object exposes a readable attribute 'lef_files'. This is the getter.

lef_files=

Signature: void **lef_files=** (string[] lef_file_paths)

Description: Sets the list technology LEF files to additionally import

See lef_files for details.

Python specific notes:
The object exposes a writable attribute 'lef_files'. This is the setter.

lef_labels_datatype

Signature: *[const]* int **lef_labels_datatype**

Description: Gets the lef_labels layer datatype value.

See produce_via_geometry for details about the layer production rules.

This method has been introduced in version 0.27.2

Python specific notes:
The object exposes a readable attribute 'lef_labels_datatype'. This is the getter.

lef_labels_datatype=

Signature: void **lef_labels_datatype=** (int datatype)

Description: Sets the lef_labels layer datatype value.

See produce_via_geometry for details about the layer production rules.

This method has been introduced in version 0.27.2

Python specific notes:
The object exposes a writable attribute 'lef_labels_datatype'. This is the setter.

lef_labels_suffix

Signature: *[const]* string **lef_labels_suffix**

Description: Gets the label layer name suffix.

See produce_via_geometry for details about the layer production rules.

This method has been introduced in version 0.27.2

Python specific notes:
The object exposes a readable attribute 'lef_labels_suffix'. This is the getter.

lef_labels_suffix=

Signature: void **lef_labels_suffix=** (string suffix)

Description: Sets the label layer name suffix.

See produce_via_geometry for details about the layer production rules.

This method has been introduced in version 0.27.2

Python specific notes:
The object exposes a writable attribute 'lef_labels_suffix'. This is the setter.

lef_pins_datatype

(1) Signature: *[const]* int **lef_pins_datatype**

Description: Gets the LEF pin geometry layer datatype value.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'lef_pins_datatype'. This is the getter.

(2) Signature: *[const]* int **lef_pins_datatype** (unsigned int mask)

Description: Gets the LEF pin geometry layer datatype value per mask.

See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...).

Mask specific rules have been introduced in version 0.27.

Python specific notes:
This method is available as 'lef_pins_datatype_' in Python to distiguish it from the property with the same name.

lef_pins_datatype=

Signature: void **lef_pins_datatype=** (int datatype)

Description: Sets the LEF pin geometry layer datatype value.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'lef_pins_datatype'. This is the setter.

lef_pins_suffix

Signature: *[const]* string **lef_pins_suffix**

Description: Gets the LEF pin geometry layer name suffix.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'lef_pins_suffix'. This is the getter.

lef_pins_suffix=

Signature: void **lef_pins_suffix=** (string suffix)

Description: Sets the LEF pin geometry layer name suffix.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'lef_pins_suffix'. This is the setter.

lef_pins_suffix_per_mask

Signature: *[const]* string **lef_pins_suffix_per_mask** (unsigned int mask)

Description: Gets the LEF pin geometry layer name suffix per mask.

See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...).

Mask specific rules have been introduced in version 0.27.

macro_layout_files

Signature: *[const]* string[] **macro_layout_files**

Description: Gets the list of layout files to read for substituting macros in DEF

These files play the same role than the macro layouts (see macro_layouts), except that this property specifies a list of file names. The given files are loaded automatically to resolve macro layouts instead of LEF geometry. See macro_resolution_mode for details when this happens. Relative paths are resolved relative to the DEF file to read or relative to the technology base path. Macros in need for substitution are looked up in the layout files by searching for cells with the same name. The files are scanned in the order they are given in the file list. The files from macro_layout_files are scanned after the layout objects specified with macro_layouts.

The setter for this property is macro_layout_files=.

This property has been added in version 0.27.1.

Python specific notes:
The object exposes a readable attribute 'macro_layout_files'. This is the getter.

macro_layout_files=

Signature: void **macro_layout_files=** (string[] file_paths)

Description: Sets the list of layout files to read for substituting macros in DEF

See macro_layout_files for details.

This property has been added in version 0.27.1.

Python specific notes:
The object exposes a writable attribute 'macro_layout_files'. This is the setter.

macro_layouts

Signature: *[const]* Layout ptr[] **macro_layouts**

Description: Gets the layout objects used for resolving LEF macros in the DEF reader.

The DEF reader can either use LEF geometry or use a separate source of layouts for the LEF macros. The macro_resolution_mode controls whether to use LEF geometry. If LEF geometry is

not used, the DEF reader will look up macro cells from the macro_layouts and pull cell layouts from there.

The LEF cells are looked up as cells by name from the macro layouts in the order these are given in this array.

macro_layout_files is another way of specifying such substitution layouts. This method accepts file names instead of layout objects.

This property has been added in version 0.27.

Python specific notes:
The object exposes a readable attribute 'macro_layouts'. This is the getter.

macro_layouts=

Signature: void **macro_layouts=** (Layout ptr[] layouts)

Description: Sets the layout objects used for resolving LEF macros in the DEF reader.

See macro_layouts for more details about this property.

Layout objects specified in the array for this property are not owned by the LEFDEFReaderConfiguration object. Be sure to keep some other reference to these Layout objects if you are storing away the LEF/DEF reader configuration object.

This property has been added in version 0.27.

Python specific notes:
The object exposes a writable attribute 'macro_layouts'. This is the setter.

macro_resolution_mode

Signature: *[const]* unsigned int **macro_resolution_mode**

Description: Gets the macro resolution mode (LEF macros into DEF).

This property describes the way LEF macros are turned into layout cells when reading DEF. There are three modes available:

- 0: produce LEF geometry unless a FOREIGN cell is specified
- 1: produce LEF geometry always and ignore FOREIGN
- 2: Never produce LEF geometry and assume FOREIGN always

If substitution layouts are specified with macro_layouts, these are used to provide macro layouts in case no LEF geometry is taken.

This property has been added in version 0.27.

Python specific notes:
The object exposes a readable attribute 'macro_resolution_mode'. This is the getter.

macro_resolution_mode=

Signature: void **macro_resolution_mode=** (unsigned int mode)

Description: Sets the macro resolution mode (LEF macros into DEF).

See macro_resolution_mode for details about this property.

This property has been added in version 0.27.

Python specific notes:
The object exposes a writable attribute 'macro_resolution_mode'. This is the setter.

map_file

Signature: *[const]* string **map_file**

Description: Gets the layer map file to use.

If a layer map file is given, the reader will pull the layer mapping from this file. The layer mapping rules specified in the reader options are ignored in this case. These are the name suffix rules for vias, blockages, routing, special routing, pins etc. and the corresponding datatype rules. The

layer_map attribute will also be ignored. The layer map file path will be resolved relative to the technology base path if the LEF/DEF reader options are used in the context of a technology.

This property has been added in version 0.27.

Python specific notes:
The object exposes a readable attribute 'map_file'. This is the getter.

map_file=

Signature: void **map_file=** (string file)

Description: Sets the layer map file to use.

See map_file for details about this property.

This property has been added in version 0.27.

Python specific notes:
The object exposes a writable attribute 'map_file'. This is the setter.

net_property_name

Signature: *[const]* variant **net_property_name**

Description: Gets a value indicating whether and how to produce net names as properties.

If set to a value not nil, net names will be attached to the net shapes generated as user properties. This attribute then specifies the user property name to be used for attaching the net names. If set to nil, no net names will be produced.

The corresponding setter is net_property_name=.

Python specific notes:
The object exposes a readable attribute 'net_property_name'. This is the getter.

net_property_name=

Signature: void **net_property_name=** (variant name)

Description: Sets a value indicating whether and how to produce net names as properties.

See net_property_name for details.

Python specific notes:
The object exposes a writable attribute 'net_property_name'. This is the setter.

new

Signature: *[static]* new LEFDEFReaderConfiguration ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

obstructions_datatype

Signature: *[const]* int **obstructions_datatype**

Description: Gets the obstruction marker layer datatype value.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'obstructions_datatype'. This is the getter.

obstructions_datatype=

Signature: void **obstructions_datatype=** (int datatype)

Description: Sets the obstruction marker layer datatype value.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'obstructions_datatype'. This is the setter.

obstructions_suffix

Signature: *[const]* string **obstructions_suffix**

Description: Gets the obstruction marker layer name suffix.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'obstructions_suffix'. This is the getter.

obstructions_suffix=

Signature: void **obstructions_suffix=** (string suffix)

Description: Sets the obstruction marker layer name suffix.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'obstructions_suffix'. This is the setter.

paths_relative_to_cwd=

Signature: void **paths_relative_to_cwd=** (bool f)

Description: Sets a value indicating whether to use paths relative to cwd (true) or DEF file (false) for map or LEF files

This write-only attribute has been introduced in version 0.27.9.

Python specific notes:
The object exposes a writable attribute 'paths_relative_to_cwd'. This is the setter.

pin_property_name

Signature: *[const]* variant **pin_property_name**

Description: Gets a value indicating whether and how to produce pin names as properties.

If set to a value not nil, pin names will be attached to the pin shapes generated as user properties. This attribute then specifies the user property name to be used for attaching the pin names. If set to nil, no pin names will be produced.

The corresponding setter is pin_property_name=.

This method has been introduced in version 0.26.4.

Python specific notes:
The object exposes a readable attribute 'pin_property_name'. This is the getter.

pin_property_name=

Signature: void **pin_property_name=** (variant name)

Description: Sets a value indicating whether and how to produce pin names as properties.

See pin_property_name for details.

This method has been introduced in version 0.26.4.

Python specific notes:
The object exposes a writable attribute 'pin_property_name'. This is the setter.

pins_datatype

(1) Signature: *[const]* int **pins_datatype**

Description: Gets the pin geometry layer datatype value.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'pins_datatype'. This is the getter.

(2) Signature: *[const]* int **pins_datatype** (unsigned int mask)

Description: Gets the pin geometry layer datatype value per mask.

See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...).

Mask specific rules have been introduced in version 0.27.

Python specific notes:

	This method is available as 'pins_datatype_' in Python to distiguish it from the property with the same name.
pins_datatype=	**Signature**: void **pins_datatype=** (int datatype) **Description**: Sets the pin geometry layer datatype value. See produce_via_geometry for details about the layer production rules. **Python specific notes:** The object exposes a writable attribute 'pins_datatype'. This is the setter.
pins_suffix	**Signature**: *[const]* string **pins_suffix** **Description**: Gets the pin geometry layer name suffix. See produce_via_geometry for details about the layer production rules. **Python specific notes:** The object exposes a readable attribute 'pins_suffix'. This is the getter.
pins_suffix=	**Signature**: void **pins_suffix=** (string suffix) **Description**: Sets the pin geometry layer name suffix. See produce_via_geometry for details about the layer production rules. **Python specific notes:** The object exposes a writable attribute 'pins_suffix'. This is the setter.
pins_suffix_per_mask	**Signature**: *[const]* string **pins_suffix_per_mask** (unsigned int mask) **Description**: Gets the pin geometry layer name suffix per mask. See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.
placement_blockage_layer	**Signature**: *[const]* string **placement_blockage_layer** **Description**: Gets the layer on which to produce the placement blockage. This attribute is a string corresponding to the string representation of LayerInfo. This string can be either a layer number, a layer/datatype pair, a name or a combination of both. See LayerInfo for details.The setter for this attribute is placement_blockage_layer=. See also produce_placement_blockages. **Python specific notes:** The object exposes a readable attribute 'placement_blockage_layer'. This is the getter.
placement_blockage_layer=	**Signature**: void **placement_blockage_layer=** (string layer) **Description**: Sets the layer on which to produce the placement blockage. See placement_blockage_layer for details. **Python specific notes:** The object exposes a writable attribute 'placement_blockage_layer'. This is the setter.
produce_blockages	**Signature**: *[const]* bool **produce_blockages** **Description**: Gets a value indicating whether routing blockage markers shall be produced. See produce_via_geometry for details about the layer production rules. **Python specific notes:**

	The object exposes a readable attribute 'produce_blockages'. This is the getter.
produce_blockages=	**Signature**: void **produce_blockages=** (bool produce) **Description**: Sets a value indicating whether routing blockage markers shall be produced. See produce_via_geometry for details about the layer production rules. **Python specific notes:** The object exposes a writable attribute 'produce_blockages'. This is the setter.
produce_cell_outlines	**Signature**: *[const]* bool **produce_cell_outlines** **Description**: Gets a value indicating whether to produce cell outlines (diearea). If set to true, cell outlines will be produced on the layer given by cell_outline_layer. **Python specific notes:** The object exposes a readable attribute 'produce_cell_outlines'. This is the getter.
produce_cell_outlines=	**Signature**: void **produce_cell_outlines=** (bool produce) **Description**: Sets a value indicating whether to produce cell outlines (diearea). See produce_cell_outlines for details. **Python specific notes:** The object exposes a writable attribute 'produce_cell_outlines'. This is the setter.
produce_fills	**Signature**: *[const]* bool **produce_fills** **Description**: Gets a value indicating whether fill geometries shall be produced. See produce_via_geometry for details about the layer production rules. Fill support has been introduced in version 0.27. **Python specific notes:** The object exposes a readable attribute 'produce_fills'. This is the getter.
produce_fills=	**Signature**: void **produce_fills=** (bool produce) **Description**: Sets a value indicating whether fill geometries shall be produced. See produce_via_geometry for details about the layer production rules. Fill support has been introduced in version 0.27. **Python specific notes:** The object exposes a writable attribute 'produce_fills'. This is the setter.
produce_labels	**Signature**: *[const]* bool **produce_labels** **Description**: Gets a value indicating whether labels shall be produced. See produce_via_geometry for details about the layer production rules. **Python specific notes:** The object exposes a readable attribute 'produce_labels'. This is the getter.
produce_labels=	**Signature**: void **produce_labels=** (bool produce) **Description**: Sets a value indicating whether labels shall be produced. See produce_via_geometry for details about the layer production rules. **Python specific notes:** The object exposes a writable attribute 'produce_labels'. This is the setter.

produce_lef_labels

Signature: *[const]* bool **produce_lef_labels**

Description: Gets a value indicating whether lef_labels shall be produced.

See produce_via_geometry for details about the layer production rules.

This method has been introduced in version 0.27.2

Python specific notes:
The object exposes a readable attribute 'produce_lef_labels'. This is the getter.

produce_lef_labels=

Signature: void **produce_lef_labels=** (bool produce)

Description: Sets a value indicating whether lef_labels shall be produced.

See produce_via_geometry for details about the layer production rules.

This method has been introduced in version 0.27.2

Python specific notes:
The object exposes a writable attribute 'produce_lef_labels'. This is the setter.

produce_lef_pins

Signature: *[const]* bool **produce_lef_pins**

Description: Gets a value indicating whether LEF pin geometries shall be produced.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'produce_lef_pins'. This is the getter.

produce_lef_pins=

Signature: void **produce_lef_pins=** (bool produce)

Description: Sets a value indicating whether LEF pin geometries shall be produced.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'produce_lef_pins'. This is the setter.

produce_obstructions

Signature: *[const]* bool **produce_obstructions**

Description: Gets a value indicating whether obstruction markers shall be produced.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'produce_obstructions'. This is the getter.

produce_obstructions=

Signature: void **produce_obstructions=** (bool produce)

Description: Sets a value indicating whether obstruction markers shall be produced.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'produce_obstructions'. This is the setter.

produce_pins

Signature: *[const]* bool **produce_pins**

Description: Gets a value indicating whether pin geometries shall be produced.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'produce_pins'. This is the getter.

produce_pins=

Signature: void **produce_pins=** (bool produce)

Description: Sets a value indicating whether pin geometries shall be produced.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'produce_pins'. This is the setter.

produce_placement_blockages

Signature: *[const]* bool **produce_placement_blockages**

Description: Gets a value indicating whether to produce placement blockage regions.

If set to true, polygons will be produced representing the placement blockage region on the layer given by placement_blockage_layer.

Python specific notes:
The object exposes a readable attribute 'produce_placement_blockages'. This is the getter.

produce_placement_blockages=

Signature: void **produce_placement_blockages=** (bool produce)

Description: Sets a value indicating whether to produce placement blockage regions.

See produce_placement_blockages for details.

Python specific notes:
The object exposes a writable attribute 'produce_placement_blockages'. This is the setter.

produce_regions

Signature: *[const]* bool **produce_regions**

Description: Gets a value indicating whether to produce regions.

If set to true, polygons will be produced representing the regions on the layer given by region_layer.

The attribute has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'produce_regions'. This is the getter.

produce_regions=

Signature: void **produce_regions=** (bool produce)

Description: Sets a value indicating whether to produce regions.

See produce_regions for details.

The attribute has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'produce_regions'. This is the setter.

produce_routing

Signature: *[const]* bool **produce_routing**

Description: Gets a value indicating whether routing geometry shall be produced.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'produce_routing'. This is the getter.

produce_routing=

Signature: void **produce_routing=** (bool produce)

Description: Sets a value indicating whether routing geometry shall be produced.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'produce_routing'. This is the setter.

produce_special_routing

Signature: *[const]* bool **produce_special_routing**

Description: Gets a value indicating whether special routing geometry shall be produced.

See produce_via_geometry for details about the layer production rules.

The differentiation between special and normal routing has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'produce_special_routing'. This is the getter.

produce_special_routing=

Signature: void **produce_special_routing=** (bool produce)

Description: Sets a value indicating whether special routing geometry shall be produced.

See produce_via_geometry for details about the layer production rules. The differentiation between special and normal routing has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'produce_special_routing'. This is the setter.

produce_via_geometry

Signature: *[const]* bool **produce_via_geometry**

Description: Sets a value indicating whether via geometries shall be produced.

If set to true, shapes will be produced for each via. The layer to be produced will be determined from the via layer's name using the suffix provided by via_geometry_suffix. If there is a specific mapping in the layer mapping table for the via layer including the suffix, the layer/datatype will be taken from the layer mapping table. If there is a mapping to the undecorated via layer, the datatype will be substituted with the via_geometry_datatype value. If no mapping is defined, a unique number will be assigned to the layer number and the datatype will be taken from the via_geometry_datatype value.

For example: the via layer is 'V1', via_geometry_suffix is 'GEO' and via_geometry_datatype is 1. Then:

- If there is a mapping for 'V1.GEO', the layer and datatype will be taken from there.
- If there is a mapping for 'V1', the layer will be taken from there and the datatype will be taken from via_geometry_datatype. The name of the produced layer will be 'V1.GEO'.
- If there is no mapping for both, the layer number will be a unique value, the datatype will be taken from via_geometry_datatype and the layer name will be 'V1.GEO'.

Python specific notes:
The object exposes a readable attribute 'produce_via_geometry'. This is the getter.

produce_via_geometry=

Signature: void **produce_via_geometry=** (bool produce)

Description: Sets a value indicating whether via geometries shall be produced.

See produce_via_geometry for details.

Python specific notes:
The object exposes a writable attribute 'produce_via_geometry'. This is the setter.

read_lef_with_def

Signature: *[const]* bool **read_lef_with_def**

Description: Gets a value indicating whether to read all LEF files in the same directory than the DEF file.

If this property is set to true (the default), the DEF reader will automatically consume all LEF files next to the DEF file in addition to the LEF files specified with lef_files. If set to false, only the LEF files specified with lef_files will be read.

This property has been added in version 0.27.

Python specific notes:
The object exposes a readable attribute 'read_lef_with_def'. This is the getter.

read_lef_with_def=

Signature: void **read_lef_with_def=** (bool flag)

Description: Sets a value indicating whether to read all LEF files in the same directory than the DEF file.

See read_lef_with_def for details about this property.

This property has been added in version 0.27.

Python specific notes:
The object exposes a writable attribute 'read_lef_with_def'. This is the setter.

region_layer

Signature: *[const]* string **region_layer**

Description: Gets the layer on which to produce the regions.

This attribute is a string corresponding to the string representation of LayerInfo. This string can be either a layer number, a layer/datatype pair, a name or a combination of both. See LayerInfo for details.The setter for this attribute is region_layer=. See also produce_regions.

The attribute has been introduced in version 0.27.

Python specific notes:
The object exposes a readable attribute 'region_layer'. This is the getter.

region_layer=

Signature: void **region_layer=** (string layer)

Description: Sets the layer on which to produce the regions.

See region_layer for details.

The attribute has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'region_layer'. This is the setter.

routing_datatype

(1) Signature: *[const]* int **routing_datatype**

Description: Gets the routing layer datatype value.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'routing_datatype'. This is the getter.

(2) Signature: *[const]* int **routing_datatype** (unsigned int mask)

Description: Gets the routing geometry layer datatype value per mask.

See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...).

Mask specific rules have been introduced in version 0.27.

Python specific notes:
This method is available as 'routing_datatype_' in Python to distiguish it from the property with the same name.

routing_datatype=

Signature: void **routing_datatype=** (int datatype)

Description: Sets the routing layer datatype value.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'routing_datatype'. This is the setter.

routing_suffix

Signature: *[const]* string **routing_suffix**

Description: Gets the routing layer name suffix.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a readable attribute 'routing_suffix'. This is the getter.

routing_suffix=

Signature: void **routing_suffix=** (string suffix)

Description: Sets the routing layer name suffix.

See produce_via_geometry for details about the layer production rules.

Python specific notes:
The object exposes a writable attribute 'routing_suffix'. This is the setter.

routing_suffix_per_mask

Signature: *[const]* string **routing_suffix_per_mask** (unsigned int mask)

Description: Gets the routing geometry layer name suffix per mask.

See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...).

Mask specific rules have been introduced in version 0.27.

separate_groups

Signature: *[const]* bool **separate_groups**

Description: Gets a value indicating whether to create separate parent cells for individual groups.

If this property is set to true, instances belonging to different groups are separated by putting them into individual parent cells. These parent cells are named after the groups and are put into the master top cell. If this property is set to false (the default), no such group parents will be formed. This property has been added in version 0.27.

Python specific notes:
The object exposes a readable attribute 'separate_groups'. This is the getter.

separate_groups=

Signature: void **separate_groups=** (bool flag)

Description: Sets a value indicating whether to create separate parent cells for individual groups.

See separate_groups for details about this property.

This property has been added in version 0.27.

Python specific notes:
The object exposes a writable attribute 'separate_groups'. This is the setter.

set_fills_datatype_per_mask

Signature: void **set_fills_datatype_per_mask** (unsigned int mask, int datatype)

Description: Sets the fill geometry layer datatype value.

See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...).

Mask specific rules have been introduced in version 0.27.

set_fills_suffix_per_mask

Signature: void **set_fills_suffix_per_mask** (unsigned int mask, string suffix)

Description: Sets the fill geometry layer name suffix per mask.

See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...).

	Mask specific rules have been introduced in version 0.27.
set_lef_pins_datatype_per_mas	**Signature**: void **set_lef_pins_datatype_per_mask** (unsigned int mask, int datatype) **Description**: Sets the LEF pin geometry layer datatype value. See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.
set_lef_pins_suffix_per_mask	**Signature**: void **set_lef_pins_suffix_per_mask** (unsigned int mask, string suffix) **Description**: Sets the LEF pin geometry layer name suffix per mask. See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.
set_pins_datatype_per_mask	**Signature**: void **set_pins_datatype_per_mask** (unsigned int mask, int datatype) **Description**: Sets the pin geometry layer datatype value. See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.
set_pins_suffix_per_mask	**Signature**: void **set_pins_suffix_per_mask** (unsigned int mask, string suffix) **Description**: Sets the pin geometry layer name suffix per mask. See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.
set_routing_datatype_per_mas	**Signature**: void **set_routing_datatype_per_mask** (unsigned int mask, int datatype) **Description**: Sets the routing geometry layer datatype value. See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.
set_routing_suffix_per_mask	**Signature**: void **set_routing_suffix_per_mask** (unsigned int mask, string suffix) **Description**: Sets the routing geometry layer name suffix per mask. See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.
set_special_routing_datatype_p	**Signature**: void **set_special_routing_datatype_per_mask** (unsigned int mask, int datatype) **Description**: Sets the special routing geometry layer datatype value. See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.
set_special_routing_suffix_per_mask	**Signature**: void **set_special_routing_suffix_per_mask** (unsigned int mask, string suffix) **Description**: Sets the special routing geometry layer name suffix per mask.

	See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.
set_via_geometry_datatype_pe	**Signature**: void **set_via_geometry_datatype_per_mask** (unsigned int mask, int datatype) **Description**: Sets the via geometry layer datatype value. See produce_via_geometry for details about this property. The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.
set_via_geometry_suffix_per_mask	**Signature**: void **set_via_geometry_suffix_per_mask** (unsigned int mask, string suffix) **Description**: Sets the via geometry layer name suffix per mask. See produce_via_geometry for details about this property. The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.
special_routing_datatype	**(1) Signature**: *[const]* int **special_routing_datatype** **Description**: Gets the special routing layer datatype value. See produce_via_geometry for details about the layer production rules. The differentiation between special and normal routing has been introduced in version 0.27. **Python specific notes:** The object exposes a readable attribute 'special_routing_datatype'. This is the getter. **(2) Signature**: *[const]* int **special_routing_datatype** (unsigned int mask) **Description**: Gets the special routing geometry layer datatype value per mask. See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27. **Python specific notes:** This method is available as 'special_routing_datatype_' in Python to distiguish it from the property with the same name.
special_routing_datatype=	**Signature**: void **special_routing_datatype=** (int datatype) **Description**: Sets the special routing layer datatype value. See produce_via_geometry for details about the layer production rules. The differentiation between special and normal routing has been introduced in version 0.27. **Python specific notes:** The object exposes a writable attribute 'special_routing_datatype'. This is the setter.
special_routing_suffix	**Signature**: *[const]* string **special_routing_suffix** **Description**: Gets the special routing layer name suffix. See produce_via_geometry for details about the layer production rules. The differentiation between special and normal routing has been introduced in version 0.27. **Python specific notes:** The object exposes a readable attribute 'special_routing_suffix'. This is the getter.

special_routing_suffix=

Signature: void **special_routing_suffix=** (string suffix)

Description: Sets the special routing layer name suffix.

See produce_via_geometry for details about the layer production rules. The differentiation between special and normal routing has been introduced in version 0.27.

Python specific notes:
The object exposes a writable attribute 'special_routing_suffix'. This is the setter.

special_routing_suffix_per_ma

Signature: *[const]* string **special_routing_suffix_per_mask** (unsigned int mask)

Description: Gets the special routing geometry layer name suffix per mask.

See produce_via_geometry for details about the layer production rules.The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...).

Mask specific rules have been introduced in version 0.27.

via_cellname_prefix

Signature: *[const]* string **via_cellname_prefix**

Description: Gets the via cellname prefix.

Vias are represented by cells. The cell name is formed by combining the via cell name prefix and the via name.

This property has been added in version 0.27.

Python specific notes:
The object exposes a readable attribute 'via_cellname_prefix'. This is the getter.

via_cellname_prefix=

Signature: void **via_cellname_prefix=** (string prefix)

Description: Sets the via cellname prefix.

See via_cellname_prefix for details about this property.

This property has been added in version 0.27.

Python specific notes:
The object exposes a writable attribute 'via_cellname_prefix'. This is the setter.

via_geometry_datatype

(1) Signature: *[const]* int **via_geometry_datatype**

Description: Gets the via geometry layer datatype value.

See produce_via_geometry for details about this property.

Python specific notes:
The object exposes a readable attribute 'via_geometry_datatype'. This is the getter.

(2) Signature: *[const]* int **via_geometry_datatype** (unsigned int mask)

Description: Gets the via geometry layer datatype value per mask.

See produce_via_geometry for details about this property. The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...).

Mask specific rules have been introduced in version 0.27.

Python specific notes:
This method is available as 'via_geometry_datatype_' in Python to distiguish it from the property with the same name.

via_geometry_datatype=

Signature: void **via_geometry_datatype=** (int datatype)

Description: Sets the via geometry layer datatype value.

See produce_via_geometry for details about this property.

	Python specific notes: The object exposes a writable attribute 'via_geometry_datatype'. This is the setter.
via_geometry_suffix	**Signature**: *[const]* string **via_geometry_suffix** **Description**: Gets the via geometry layer name suffix. See produce_via_geometry for details about this property. **Python specific notes:** The object exposes a readable attribute 'via_geometry_suffix'. This is the getter.
via_geometry_suffix=	**Signature**: void **via_geometry_suffix=** (string suffix) **Description**: Sets the via geometry layer name suffix. See produce_via_geometry for details about this property. **Python specific notes:** The object exposes a writable attribute 'via_geometry_suffix'. This is the setter.
via_geometry_suffix_per_mask	**Signature**: *[const]* string **via_geometry_suffix_per_mask** (unsigned int mask) **Description**: Gets the via geometry layer name suffix per mask. See produce_via_geometry for details about this property. The mask number is a zero-based mask index (0: MASK 1, 1: MASK 2 ...). Mask specific rules have been introduced in version 0.27.

4.235. API reference - Class NetTracerConnectionInfo

Notation used in Ruby API documentation

Module: db

Description: Represents a single connection info line for the net tracer technology definition

This class has been introduced in version 0.28.3.

Public constructors

new NetTracerConnectionInfo ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetTracer(other)	Assigns another object to self
[const]	new NetTracerConnectionInfo ptr	**dup**		Creates a copy of self
[const]	string	**layer_a**		Gets the expression for the A layer
[const]	string	**layer_b**		Gets the expression for the B layer
[const]	string	**via_layer**		Gets the expression for the Via layer

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const NetTracerConnectionInfo other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetTracerConnectionInfo ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

layer_a

Signature: *[const]* string **layer_a**

Description: Gets the expression for the A layer

layer_b

Signature: *[const]* string **layer_b**

Description: Gets the expression for the B layer

new

Signature: *[static]* new NetTracerConnectionInfo ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

via_layer

Signature: *[const]* string **via_layer**

Description: Gets the expression for the Via layer

4.236. API reference - Class NetTracerSymbolInfo

Notation used in Ruby API documentation

Module: db

Description: Represents a single symbol info line for the net tracer technology definition

This class has been introduced in version 0.28.3.

Public constructors

new NetTracerSymbolInfo ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetTracer other)	Assigns another object to self
[const]	new NetTracerSymbolInfo ptr	**dup**		Creates a copy of self
[const]	string	**expression**		Gets the expression
[const]	string	**symbol**		Gets the symbol

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const NetTracerSymbolInfo other)

Description: Assigns another object to self

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetTracerSymbolInfo ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

expression

Signature: *[const]* string **expression**

Description: Gets the expression

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

new

Signature: *[static]* new NetTracerSymbolInfo ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

symbol

Signature: *[const]* string **symbol**

Description: Gets the symbol

4.237. API reference - Class NetTracerConnectivity

Notation used in Ruby API documentation

Module: db

Description: A connectivity description for the net tracer

This object represents the technology description for the net tracer (represented by the NetTracer class). A technology description basically consists of connection declarations. A connection is given by either two or three expressions describing two conductive materials. With two expressions, the connection describes a transition from one material to another one. With three expressions, the connection describes a transition from one material to another through a connection (a "via").

The conductive material is derived from original layers either directly or through boolean expressions. These expressions can include symbols which are defined through the symbol method.

For details about the expressions see the description of the net tracer feature.

This class has been introduced in version 0.28 and replaces the 'NetTracerTechnology' class which has been generalized.

Public constructors

new NetTracerConnectivity ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetTracerConne other)	Assigns another object to self
	void	**connection**	(string a, string b)	Defines a connection between two materials
	void	**connection**	(string a, string via, string b)	Defines a connection between materials through a via
[const]	string	**description**		Gets the description text of the connectivty definition
	void	**description=**	(string d)	Sets the description of the connectivty definition
[const]	new NetTracerConnectivity ptr	**dup**		Creates a copy of self

[const,iter]	NetTracerConnectionIn	**each_connection**		Gets the connection information.
[const,iter]	NetTracerSymbolInfo	**each_symbol**		Gets the symbol information.
[const]	string	**name**		Gets the name of the connectivty definition
	void	**name=**	(string n)	Sets the name of the connectivty definition
	void	**symbol**	(string name, string expr)	Defines a symbol for use in the material expressions.

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is

	known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required. Usually it's not required to call this method. It has been introduced in version 0.24.
_unmanage	**Signature**: void **_unmanage** **Description**: Marks the object as no longer owned by the script side. Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur. Usually it's not required to call this method. It has been introduced in version 0.24.
assign	**Signature**: void **assign** (const NetTracerConnectivity other) **Description**: Assigns another object to self
connection	**(1) Signature**: void **connection** (string a, string b) **Description**: Defines a connection between two materials See the class description for details about this method. **(2) Signature**: void **connection** (string a, string via, string b) **Description**: Defines a connection between materials through a via See the class description for details about this method.
create	**Signature**: void **create** **Description**: Ensures the C++ object is created Use of this method is deprecated. Use _create instead Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.
description	**Signature**: *[const]* string **description** **Description**: Gets the description text of the connectivty definition The description is an optional string giving a human-readable description for this definition. **Python specific notes:** The object exposes a readable attribute 'description'. This is the getter.
description=	**Signature**: void **description=** (string d) **Description**: Sets the description of the connectivty definition **Python specific notes:** The object exposes a writable attribute 'description'. This is the setter.
destroy	**Signature**: void **destroy** **Description**: Explicitly destroys the object Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetTracerConnectivity ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_connection

Signature: *[const,iter]* NetTracerConnectionInfo **each_connection**

Description: Gets the connection information.

This iterator method has been introduced in version 0.28.3.

each_symbol

Signature: *[const,iter]* NetTracerSymbolInfo **each_symbol**

Description: Gets the symbol information.

This iterator method has been introduced in version 0.28.3.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

name

Signature: *[const]* string **name**

Description: Gets the name of the connectivty definition

The name is an optional string defining the formal name for this definition.

Python specific notes:
The object exposes a readable attribute 'name'. This is the getter.

name=

Signature: void **name=** (string n)

Description: Sets the name of the connectivty definition

Python specific notes:
The object exposes a writable attribute 'name'. This is the setter.

new

Signature: *[static]* new NetTracerConnectivity ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

symbol

Signature: void **symbol** (string name, string expr)

Description: Defines a symbol for use in the material expressions.

Defines a sub-expression to be used in further symbols or material expressions. For the detailed notation of the expression see the description of the net tracer feature.

4.238. API reference - Class NetTracerTechnologyComponent

Notation used in Ruby API documentation

Module: db

Description: Represents the technology information for the net tracer.

Class hierarchy: NetTracerTechnologyComponent » TechnologyComponent

This class has been redefined in version 0.28 and re-introduced in version 0.28.3. Since version 0.28, multiple stacks are supported and the individual stack definition is provided through a list of stacks. Use each to iterate the stacks.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**add**	(const NetTracerConn connection)	Adds a connectivity definition.
	void	**assign**	(const NetTracerTechnologyComponent other)	Assigns another object to self
	void	**clear**		Removes all connectivity definitions.
[const]	new NetTracerTechnologyComponent ptr	**dup**		Creates a copy of self
[const,iter]	NetTracerConnectivity	**each**		Gets the connectivity definitions from the net tracer technology component.

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

add

Signature: void **add** (const NetTracerConnectivity connection)

Description: Adds a connectivity definition.

This method has been introduced in version 0.28.7

assign

Signature: void **assign** (const NetTracerTechnologyComponent other)

Description: Assigns another object to self

clear

Signature: void **clear**

Description: Removes all connectivity definitions.

This method has been introduced in version 0.28.7

dup

Signature: *[const]* new NetTracerTechnologyComponent ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each

Signature: *[const,iter]* NetTracerConnectivity **each**

Description: Gets the connectivity definitions from the net tracer technology component.

Python specific notes:
This method enables iteration of the object.

4.239. API reference - Class NetElement

Notation used in Ruby API documentation

Module: db

Description: A net element for the NetTracer net tracing facility

This object represents a piece of a net extracted by the net tracer. See the description of NetTracer for more details about the net tracer feature.

The NetTracer object represents one shape of the net. The shape can be an original shape or a shape derived in a boolean operation. In the first case, the shape refers to a shape within a cell or a subcell of the original top cell. In the latter case, the shape is a synthesized one and outside the original layout hierarchy.

In any case, the shape method will deliver the shape and trans the transformation of the shape into the original top cell. To obtain a flat representation of the net, the shapes need to be transformed by this transformation.

layer will give the layer the shape is located at, cell_index will denote the cell that contains the shape.

This class has been introduced in version 0.25.

Public constructors

new NetElement ptr	**new**		Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetElem other)	Assigns another object to self
[const]	Box	**bbox**		Delivers the bounding box of the shape as seen from the original top cell
[const]	unsigned int	**cell_index**		Gets the index of the cell the shape is inside
[const]	new NetElement ptr	**dup**		Creates a copy of self
[const]	unsigned int	**layer**		Gets the index of the layer the shape is on
[const]	Shape	**shape**		Gets the shape that makes up this net element
[const]	ICplxTrans	**trans**		Gets the transformation to apply for rendering the shape in the original top cell

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void _**create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void _**destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool _**destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool _**is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void _**manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void _**unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method

will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const NetElement other)

Description: Assigns another object to self

bbox

Signature: *[const]* Box **bbox**

Description: Delivers the bounding box of the shape as seen from the original top cell

cell_index

Signature: *[const]* unsigned int **cell_index**

Description: Gets the index of the cell the shape is inside

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetElement ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

layer	**Signature**: *[const]* unsigned int **layer** **Description**: Gets the index of the layer the shape is on
new	**Signature**: *[static]* new NetElement ptr **new** **Description**: Creates a new object of this class **Python specific notes:** This method is the default initializer of the object.
shape	**Signature**: *[const]* Shape **shape** **Description**: Gets the shape that makes up this net element See the class description for more details about this attribute.
trans	**Signature**: *[const]* ICplxTrans **trans** **Description**: Gets the transformation to apply for rendering the shape in the original top cell See the class description for more details about this attribute.

4.240. API reference - Class NetTracer

Notation used in Ruby API documentation

Module: db

Description: The net tracer feature

The net tracer class provides an interface to the net tracer feature. It is accompanied by the NetElement and NetTracerTechnology classes. The latter will provide the technology definition for the net tracer while the NetElement objects represent a piece of the net after it has been extracted.

The technology definition is optional. The net tracer can be used with a predefined technology as well. The basic scheme of using the net tracer is to instantiate a net tracer object and run the extraction through the NetTracer#trace method. After this method was executed successfully, the resulting net can be obtained from the net tracer object by iterating over the NetElement objects of the net tracer.

Here is some sample code:

```
ly = RBA::CellView::active.layout

tracer = RBA::NetTracer::new

tech = RBA::NetTracerConnectivity::new
tech.connection("1/0", "2/0", "3/0")

tracer.trace(tech, ly, ly.top_cell, RBA::Point::new(7000, 1500), ly.find_layer(1, 0))

tracer.each_element do |e|
  puts e.shape.polygon.transformed(e.trans)
end
```

This class has been introduced in version 0.25. With version 0.28, the NetTracerConnectivity class replaces the 'NetTracerTechnology' class.

Public constructors

new NetTracer ptr	**new**	Creates a new object of this class

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**assign**	(const NetTracer other)	Assigns another object to self
	void	**clear**		Clears the data from the last extraction

[const]	new NetTracer ptr	**dup**		Creates a copy of self
[const,iter]	NetElement	**each_element**		Iterates over the elements found during extraction
[const]	bool	**incomplete?**		Returns a value indicating whether the net is incomplete
[const]	string	**name**		Returns the name of the net found during extraction
[const]	unsigned long	**num_elements**		Returns the number of elements found during extraction
	void	**trace**	(const NetTracerConnectivity tech, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer)	Runs a net extraction
	void	**trace**	(const NetTracerConnectivity tech, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer, const Point stop_point, unsigned int stop_layer)	Runs a path extraction
	void	**trace**	(string tech, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer)	Runs a net extraction taking a predefined technology
	void	**trace**	(string tech, string connectivity_name, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer)	Runs a net extraction taking a predefined technology
	void	**trace**	(string tech, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer, const Point stop_point, unsigned int stop_layer)	Runs a path extraction taking a predefined technology
	void	**trace**	(string tech, string connectivity_name, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer, const Point stop_point,	Runs a path extraction taking a predefined technology

			unsigned int stop_layer)	
[const]	unsigned long	**trace_depth**		gets the trace depth
	void	**trace_depth=**	(unsigned long n)	Sets the trace depth (shape limit)

Deprecated methods (protected, public, static, non-static and constructors)

	void	**create**	Use of this method is deprecated. Use _create instead
	void	**destroy**	Use of this method is deprecated. Use _destroy instead
[const]	bool	**destroyed?**	Use of this method is deprecated. Use _destroyed? instead
[const]	bool	**is_const_object?**	Use of this method is deprecated. Use _is_const_object? instead

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

assign

Signature: void **assign** (const NetTracer other)

Description: Assigns another object to self

clear

Signature: void **clear**

Description: Clears the data from the last extraction

create

Signature: void **create**

Description: Ensures the C++ object is created

Use of this method is deprecated. Use _create instead

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

destroy

Signature: void **destroy**

Description: Explicitly destroys the object

Use of this method is deprecated. Use _destroy instead

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

destroyed?

Signature: *[const]* bool **destroyed?**

Description: Returns a value indicating whether the object was already destroyed

Use of this method is deprecated. Use _destroyed? instead

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

dup

Signature: *[const]* new NetTracer ptr **dup**

Description: Creates a copy of self

Python specific notes:
This method also implements '__copy__' and '__deepcopy__'.

each_element

Signature: *[const,iter]* NetElement **each_element**

Description: Iterates over the elements found during extraction

The elements are available only after the extraction has been performed.

incomplete?

Signature: *[const]* bool **incomplete?**

Description: Returns a value indicating whether the net is incomplete

A net may be incomplete if the extraction has been stopped by the user for example. This attribute is useful only after the extraction has been performed.

is_const_object?

Signature: *[const]* bool **is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

Use of this method is deprecated. Use _is_const_object? instead

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

name

Signature: *[const]* string **name**

Description: Returns the name of the net found during extraction

The net name is extracted from labels found during the extraction. This attribute is useful only after the extraction has been performed.

new

Signature: *[static]* new NetTracer ptr **new**

Description: Creates a new object of this class

Python specific notes:
This method is the default initializer of the object.

num_elements

Signature: *[const]* unsigned long **num_elements**

Description: Returns the number of elements found during extraction

This attribute is useful only after the extraction has been performed.

trace

(1) Signature: void **trace** (const NetTracerConnectivity tech, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer)

Description: Runs a net extraction

tech:	The connectivity definition
layout:	The layout on which to run the extraction
cell:	The cell on which to run the extraction (child cells will be included)
start_point:	The start point from which to start extraction of the net
start_layer:	The layer from which to start extraction

This method runs an extraction with the given parameters. To make the extraction successful, a shape must be present at the given start point on the start layer. The start layer must be a valid layer mentioned within the technology specification.

This version runs a single extraction - i.e. it will extract all elements connected to the given seed point. A path extraction version is provided as well which will extract one (the presumably shortest) path between two points.

(2) Signature: void **trace** (const NetTracerConnectivity tech, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer, const Point stop_point, unsigned int stop_layer)

Description: Runs a path extraction

tech:	The connectivity definition
layout:	The layout on which to run the extraction
cell:	The cell on which to run the extraction (child cells will be included)

start_point:	The start point from which to start extraction of the net
start_layer:	The layer from which to start extraction
stop_point:	The stop point at which to stop extraction of the net
stop_layer:	The layer at which to stop extraction

This method runs an path extraction with the given parameters. To make the extraction successful, a shape must be present at the given start point on the start layer and at the given stop point at the given stop layer. The start and stop layers must be a valid layers mentioned within the technology specification.

This version runs a path extraction and will deliver elements forming one path leading from the start to the end point.

(3) Signature: void **trace** (string tech, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer)

Description: Runs a net extraction taking a predefined technology

This method behaves identical as the version with a technology object, except that it will look for a technology with the given name to obtain the extraction setup. The technology is looked up by technology name. A version of this method exists where it is possible to specify the name of the particular connectivity to use in case there are multiple definitions available.

(4) Signature: void **trace** (string tech, string connectivity_name, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer)

Description: Runs a net extraction taking a predefined technology

This method behaves identical as the version with a technology object, except that it will look for a technology with the given name to obtain the extraction setup. This version allows specifying the name of the connecvitiy setup.

This method variant has been introduced in version 0.28.

(5) Signature: void **trace** (string tech, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer, const Point stop_point, unsigned int stop_layer)

Description: Runs a path extraction taking a predefined technology

This method behaves identical as the version with a technology object, except that it will look for a technology with the given name to obtain the extraction setup.

(6) Signature: void **trace** (string tech, string connectivity_name, const Layout layout, const Cell cell, const Point start_point, unsigned int start_layer, const Point stop_point, unsigned int stop_layer)

Description: Runs a path extraction taking a predefined technology

This method behaves identical as the version with a technology object, except that it will look for a technology with the given name to obtain the extraction setup.This version allows specifying the name of the connecvitiy setup.

This method variant has been introduced in version 0.28.

trace_depth

Signature: *[const]* unsigned long **trace_depth**

Description: gets the trace depth

See trace_depth= for a description of this property.

This method has been introduced in version 0.26.4.

Python specific notes:
The object exposes a readable attribute 'trace_depth'. This is the getter.

trace_depth=

Signature: void **trace_depth=** (unsigned long n)

Description: Sets the trace depth (shape limit)

Set this value to limit the maximum number of shapes delivered. Upon reaching this count, the tracer will stop and report the net as 'incomplete' (see incomplete?). Setting a trace depth if 0 is equivalent to 'unlimited'. The actual number of shapes delivered may be a little less than the depth because of internal marker shapes which are taken into account, but are not delivered.

This method has been introduced in version 0.26.4.

Python specific notes:
The object exposes a writable attribute 'trace_depth'. This is the setter.

4.241. API reference - Class D25View

Notation used in Ruby API documentation

Module: lay

Description: The 2.5d View Dialog

Class hierarchy: D25View » QDialog » QWidget » QObject

This class is used internally to implement the 2.5d feature.

This class has been introduced in version 0.28.

Public methods

	void	**_create**		Ensures the C++ object is created
	void	**_destroy**		Explicitly destroys the object
[const]	bool	**_destroyed?**		Returns a value indicating whether the object was already destroyed
[const]	bool	**_is_const_object?**		Returns a value indicating whether the reference is a const reference
	void	**_manage**		Marks the object as managed by the script side.
	void	**_unmanage**		Marks the object as no longer owned by the script side.
	void	**begin**	(string generator)	Initiates delivery of display groups
	void	**clear**		Clears all display entries in the view
	void	**close**		Closes the view
	void	**close_display**		Finishes the display group
	void	**entry**	(const Region data, double dbu, double zstart, double zstop)	Creates a new display entry in the group opened with open_display
	void	**entry**	(const Edges data, double dbu, double zstart, double zstop)	Creates a new display entry in the group opened with open_display
	void	**entry**	(const EdgePairs data, double dbu, double zstart, double zstop)	Creates a new display entry in the group opened with open_display
	void	**finish**		Finishes the view - call this after the display groups have been created
	void	**open_display**	(unsigned int ptr frame_color,	Creates a new display group

unsigned int ptr
fill_color,
const LayerInfo ptr
like,
string ptr name)

Detailed description

_create

Signature: void **_create**

Description: Ensures the C++ object is created

Use this method to ensure the C++ object is created, for example to ensure that resources are allocated. Usually C++ objects are created on demand and not necessarily when the script object is created.

_destroy

Signature: void **_destroy**

Description: Explicitly destroys the object

Explicitly destroys the object on C++ side if it was owned by the script interpreter. Subsequent access to this object will throw an exception. If the object is not owned by the script, this method will do nothing.

_destroyed?

Signature: *[const]* bool **_destroyed?**

Description: Returns a value indicating whether the object was already destroyed

This method returns true, if the object was destroyed, either explicitly or by the C++ side. The latter may happen, if the object is owned by a C++ object which got destroyed itself.

_is_const_object?

Signature: *[const]* bool **_is_const_object?**

Description: Returns a value indicating whether the reference is a const reference

This method returns true, if self is a const reference. In that case, only const methods may be called on self.

_manage

Signature: void **_manage**

Description: Marks the object as managed by the script side.

After calling this method on an object, the script side will be responsible for the management of the object. This method may be called if an object is returned from a C++ function and the object is known not to be owned by any C++ instance. If necessary, the script side may delete the object if the script's reference is no longer required.

Usually it's not required to call this method. It has been introduced in version 0.24.

_unmanage

Signature: void **_unmanage**

Description: Marks the object as no longer owned by the script side.

Calling this method will make this object no longer owned by the script's memory management. Instead, the object must be managed in some other way. Usually this method may be called if it is known that some C++ object holds and manages this object. Technically speaking, this method will turn the script's reference into a weak reference. After the script engine decides to delete the reference, the object itself will still exist. If the object is not managed otherwise, memory leaks will occur.

Usually it's not required to call this method. It has been introduced in version 0.24.

begin	**Signature**: void **begin** (string generator) **Description**: Initiates delivery of display groups
clear	**Signature**: void **clear** **Description**: Clears all display entries in the view
close	**Signature**: void **close** **Description**: Closes the view
close_display	**Signature**: void **close_display** **Description**: Finishes the display group
entry	**(1) Signature**: void **entry** (const Region data, double dbu, double zstart, double zstop) **Description**: Creates a new display entry in the group opened with open_display **(2) Signature**: void **entry** (const Edges data, double dbu, double zstart, double zstop) **Description**: Creates a new display entry in the group opened with open_display **(3) Signature**: void **entry** (const EdgePairs data, double dbu, double zstart, double zstop) **Description**: Creates a new display entry in the group opened with open_display
finish	**Signature**: void **finish** **Description**: Finishes the view - call this after the display groups have been created
open_display	**Signature**: void **open_display** (unsigned int ptr frame_color, unsigned int ptr fill_color, const LayerInfo ptr like, string ptr name) **Description**: Creates a new display group

4.242. API reference - Class PCellDeclarationHelper

Notation used in Ruby API documentation

Module: db

Description: A helper class to simplify the declaration of a PCell (Ruby version)

Class hierarchy: PCellDeclarationHelper » PCellDeclaration

This class provides adds some convenience to the PCell declaration based on PCellDeclaration. PCellDeclaration is a C++ object which is less convenient to use than a Ruby-based approach. In particular this class simplifies the declaration and use of parameters through accessor methods that are created automatically from the declaration of the parameters.

The basic usage of this class is the following:

```
# Derive your PCell from PCellDeclarationHelper
class MyPCell < RBA::PCellDeclarationHelper

  # initalize
  def initialize
    super
    # your initialization: add parameters with name, type, description and
    # optional other values
    param :p, TypeInt, "The parameter", :default => 1
    param :l, TypeLayer, "The layer", :default => RBA::LayerInfo::new(1, 0)
    # add other parameters ..
  end

  # reimplement display_text_impl
  def display_text_impl
    # implement the method here
  end

  # reimplement produce_impl
  def produce_impl
    # implement the method here
  end

  # optionally reimplement coerce_parameters_impl
  def coerce_parameters_impl
    # implement the method here
  end

end
```

An implementation of display_text_impl could look like this:

```
  def display_text_impl
    "We have p=#{p}"
  end
```

Because in the sample declaration above we have declared parameter "p" we can access the value of p inside the implementation simply by using the "p" method.

Similarily the produce_impl implementation could use code like the following. Please note that layout and cell are available to get the layout and cell. Also because we have declared a layer parameter "l", we can access the layer index with the "l_layer" method:

```
  def produce_impl
    cell.shapes(l_layer).insert(RBA::Box.new(0, 0, p*100, p*200))
```

```
end
```

Again in this sample, we used "p" to access the parameter "p".

The implementation of coerce_parameter_impl can make use of the parameter setters. In the case of the "p" parameter, the setter is "set_p":

```
def coerce_parameter_impl
  p < 10 || set_p(10)
end
```

Public methods

callback_impl		Provides a callback on a parameter change
callback_impl		Provides a callback on a parameter change
can_create_from_shape_impl		Returns true if the PCell can be created from the given shape
can_create_from_shape_impl		Returns true if the PCell can be created from the given shape
cell		Gets the reference to the current cell within produce_impl
cell		Gets the reference to the current cell within produce_impl
coerce_parameters_impl		Coerces the parameters
coerce_parameters_impl		Coerces the parameters
display_text_impl		Delivers the display text
display_text_impl		Delivers the display text
initialize		Initializes this instance
initialize		Initializes this instance
layer		Gets the reference to the current layer index within can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl
layer		Gets the reference to the current layer index within can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl
layout		Gets the reference to the current layout within produce_impl, can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl
layout		Gets the reference to the current layout within produce_impl, can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl
param	(name, type, descriptio	Declares a parameter with the given name, type and description and optional attributes.

	...)	
param	(name, type, description, ...)	Declares a parameter with the given name, type and description and optional attributes.
parameters_from_shape_impl		Sets the parameters from a shape
parameters_from_shape_impl		Sets the parameters from a shape
produce_impl		Produces the layout
produce_impl		Produces the layout
shape		Gets the reference to the current shape within can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl
shape		Gets the reference to the current shape within can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl
transformation_from_shape_imp		Gets the initial PCell instance transformation when creating from a shape
transformation_from_shape_impl		Gets the initial PCell instance transformation when creating from a shape

Detailed description

callback_impl

(1) Signature: **callback_impl**

Description: Provides a callback on a parameter change

This method applies to user interface changes only. Whenever a parameter is changed on the parameter page, this method is called with the name of the parameter.

On some occasions, this method called to establish a configuration unspecifically. In this case, the name is an empty string - indicating "all parameters may have changed".

This method can change the state of this or any other parameter. For this, the state objects are supplied instead of the parameter values. For example to enable parameter "b" when a boolean parameter "a" is true, use the following code:

```
def callback_impl(name)
  if name == "a" || name == ""
    b.enabled = a.value
  end
end
```

The "enabled" attribute of the PCellParameterState object indicates whether the parameter is enabled in the user interface. "a.value" delivers the value of the (boolean type assumed) parameter "a".

Note that the above code also checks for empty name to consider the case of a global refresh.

Further useful attributes of the parameters are:

- **enabled** : the parameter entry is grayed out if false

- **readonly** : the parameter cannot be edited (less strict than enabled)
- **visible** : the parameter entry is not visible if false
- **icon** : Sets an icon in front of the parameter indicating an error or a warning (use PCellParameterState#WarningIcon or PCellParameterState#ErrorIcon).

(2) Signature: **callback_impl**

Description: Provides a callback on a parameter change

This method applies to user interface changes only. Whenever a parameter is changed on the parameter page, this method is called with the name of the parameter.

On some occasions, this method called to establish a configuration unspecifically. In this case, the name is an empty string - indicating "all parameters may have changed".

This method can change the state of this or any other parameter. For this, the state objects are supplied instead of the parameter values. For example to enable parameter "b" when a boolean parameter "a" is true, use the following code:

```
def callback_impl(name)
  if name == "a" || name == ""
    b.enabled = a.value
  end
end
```

The "enabled" attribute of the PCellParameterState object indicates whether the parameter is enabled in the user interface. "a.value" delivers the value of the (boolean type assumed) parameter "a".

Note that the above code also checks for empty name to consider the case of a global refresh.

Further useful attributes of the parameters are:

- **enabled** : the parameter entry is grayed out if false
- **readonly** : the parameter cannot be edited (less strict than enabled)
- **visible** : the parameter entry is not visible if false
- **icon** : Sets an icon in front of the parameter indicating an error or a warning (use PCellParameterState#WarningIcon or PCellParameterState#ErrorIcon).

can_create_from_shape_impl

(1) Signature: **can_create_from_shape_impl**

Description: Returns true if the PCell can be created from the given shape

This method can be reimplemented in a PCell class. If the PCell can be created from the shape available through the shape accessor (a Shape object), this method is supposed to return true. The layout the shape lives in can be accessed with layout and the layer with layer.

The default implementation returns false.

(2) Signature: **can_create_from_shape_impl**

Description: Returns true if the PCell can be created from the given shape

This method can be reimplemented in a PCell class. If the PCell can be created from the shape available through the shape accessor (a Shape object), this method is supposed to return true. The layout the shape lives in can be accessed with layout and the layer with layer.

The default implementation returns false.

cell

(1) Signature: **cell**

Description: Gets the reference to the current cell within produce_impl

(2) Signature: **cell**

Description: Gets the reference to the current cell within produce_impl

coerce_parameters_impl

(1) Signature: **coerce_parameters_impl**

Description: Coerces the parameters

This method can be reimplemented in a PCell class. It is supposed to adjust parameters to render a consistent parameter set and to fix parameter range errors. This method is called for example inside the PCell user interface to compute the actual parameters when "Apply" is pressed.

(2) Signature: **coerce_parameters_impl**

Description: Coerces the parameters

This method can be reimplemented in a PCell class. It is supposed to adjust parameters to render a consistent parameter set and to fix parameter range errors. This method is called for example inside the PCell user interface to compute the actual parameters when "Apply" is pressed.

display_text_impl

(1) Signature: **display_text_impl**

Description: Delivers the display text

This method must be reimplemented in a PCell class to identify the PCell in human-readable form. This text is shown in the cell tree for the PCell for example.

(2) Signature: **display_text_impl**

Description: Delivers the display text

This method must be reimplemented in a PCell class to identify the PCell in human-readable form. This text is shown in the cell tree for the PCell for example.

initialize

(1) Signature: **initialize**

Description: Initializes this instance

(2) Signature: **initialize**

Description: Initializes this instance

layer

(1) Signature: **layer**

Description: Gets the reference to the current layer index within can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl

The object returned is the layer index within the Layout object of the shape which will be converted.

(2) Signature: **layer**

Description: Gets the reference to the current layer index within can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl

The object returned is the layer index within the Layout object of the shape which will be converted.

layout

(1) Signature: **layout**

Description: Gets the reference to the current layout within produce_impl, can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl

The object returned is the Layout object of the shape which will be converted.

(2) Signature: **layout**

Description: Gets the reference to the current layout within produce_impl, can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl

The object returned is the Layout object of the shape which will be converted.

param

(1) Signature: **param** (name, type, description, ...)

Description: Declares a parameter with the given name, type and description and optional attributes.

name:	The name of the parameter. Must be a simple word.
type:	The type. One of the Type... constants, that this class borrowed from PCellParameterDeclaration.
description:	The description text for this parameter

Optional, named parameters are

- **:hidden**: (boolean) true, if the parameter is not shown in the dialog
- **:readonly**: (boolean) true, if the parameter cannot be edited
- **:unit**: the unit string
- **:min_value**: the minimum value (effective for numerical types and if no choices are present)
- **:max_value**: the maximum value (effective for numerical types and if no choices are present)
- **:default**: the default value
- **:choices**: ([[d, v], ...]) choice descriptions/value for choice type

":choices" must be an array of two-element arrays (description text, value) which specify one choice each for parameters with a choice of values. Such parameters are represented by a drop-down box.

This declaration will create accessor methods "x" and "set_x", where "x" is the name of the parameter. If the type is TypeLayer, an accessor "x_layer" delivering the layer index inside produce_impl is created as well.

(2) Signature: **param** (name, type, description, ...)

Description: Declares a parameter with the given name, type and description and optional attributes.

name:	The name of the parameter. Must be a simple word.
type:	The type. One of the Type... constants, that this class borrowed from PCellParameterDeclaration.
description:	The description text for this parameter

Optional, named parameters are

- **:hidden**: (boolean) true, if the parameter is not shown in the dialog
- **:readonly**: (boolean) true, if the parameter cannot be edited
- **:unit**: the unit string
- **:min_value**: the minimum value (effective for numerical types and if no choices are present)
- **:max_value**: the maximum value (effective for numerical types and if no choices are present)
- **:default**: the default value
- **:choices**: ([[d, v], ...]) choice descriptions/value for choice type

":choices" must be an array of two-element arrays (description text, value) which specify one choice each for parameters with a choice of values. Such parameters are represented by a drop-down box.

This declaration will create accessor methods "x" and "set_x", where "x" is the name of the parameter. If the type is TypeLayer, an accessor "x_layer" delivering the layer index inside produce_impl is created as well.

parameters_from_shape_impl

(1) Signature: **parameters_from_shape_impl**

Description: Sets the parameters from a shape

This method can be reimplemented in a PCell class. If can_create_from_shape_impl returns true, this method is called to set the parameters from the given shape (see shape, layout and layer). Note, that for setting a layer parameter you need to create the LayerInfo object, i.e. like this:

```
set_l layout.get_info(layer)
```

The default implementation does nothing. All parameters not set in this method will receive their default value.

If you use a parameter called "layer" for example, the parameter getter will hide the "layer" argument. Use "_layer" for the argument in this case (same for "layout", "shape" or "cell):

```
set_layer layout.get_info(_layer)
```

(2) Signature: **parameters_from_shape_impl**

Description: Sets the parameters from a shape

This method can be reimplemented in a PCell class. If can_create_from_shape_impl returns true, this method is called to set the parameters from the given shape (see shape, layout and layer). Note, that for setting a layer parameter you need to create the LayerInfo object, i.e. like this:

```
set_l layout.get_info(layer)
```

The default implementation does nothing. All parameters not set in this method will receive their default value.

If you use a parameter called "layer" for example, the parameter getter will hide the "layer" argument. Use "_layer" for the argument in this case (same for "layout", "shape" or "cell):

```
set_layer layout.get_info(_layer)
```

produce_impl

(1) Signature: **produce_impl**

Description: Produces the layout

This method must be reimplemented in a PCell class. Using the parameter values provided by the parameter accessor methods and the layout and cell through layout and cell, this method is supposed to produce the final layout inside the given cell.

(2) Signature: **produce_impl**

Description: Produces the layout

This method must be reimplemented in a PCell class. Using the parameter values provided by the parameter accessor methods and the layout and cell through layout and cell, this method is supposed to produce the final layout inside the given cell.

shape

(1) Signature: **shape**

Description: Gets the reference to the current shape within can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl

The object returned is the Shape object of the shape which will be converted.

(2) Signature: **shape**

Description: Gets the reference to the current shape within can_create_from_shape_impl, parameters_from_shape_impl and transformation_from_shape_impl

The object returned is the Shape object of the shape which will be converted.

transformation_from_shape_impl

(1) Signature: **transformation_from_shape_impl**

Description: Gets the initial PCell instance transformation when creating from a shape

This method can be reimplemented in a PCell class. If can_create_from_shape_impl returns true, this method is called to get the initial transformation from the given shape (see shape, layout and layer).

This method must return a Trans object. The default implementation returns a unit transformation (no displacement, no rotation).

(2) Signature: **transformation_from_shape_impl**

Description: Gets the initial PCell instance transformation when creating from a shape

This method can be reimplemented in a PCell class. If can_create_from_shape_impl returns true, this method is called to get the initial transformation from the given shape (see shape, layout and layer).

This method must return a Trans object. The default implementation returns a unit transformation (no displacement, no rotation).

4.243. Class Index for Module db

KLayout classes

Box	A box class with integer coordinates
Cell	A cell
CellInstArray	A single or array cell instance
CellMapping	A cell mapping (source to target layout)
Circuit	Circuits are the basic building blocks of the netlist
CompoundRegionOperationNode	A base class for compound DRC operations
CompoundRegionOperationNode::GeometricalOp	This class represents the CompoundRegionOperationNode::GeometricalOp enum
CompoundRegionOperationNode::LogicalOp	This class represents the CompoundRegionOperationNode::LogicalOp enum
CompoundRegionOperationNode::ParameterType	This class represents the parameter type enum used in \CompoundRegionOperationNode#new_bbox_filter
CompoundRegionOperationNode::RatioParameterType	This class represents the parameter type enum used in \CompoundRegionOperationNode#new_ratio_filter
CompoundRegionOperationNode::ResultType	This class represents the CompoundRegionOperationNode::ResultType enum
Connectivity	This class specifies connections between different layers.
CplxTrans	A complex transformation
DBox	A box class with floating-point coordinates
DCellInstArray	A single or array cell instance in micrometer units
DCplxTrans	A complex transformation
DEdge	An edge class
DEdgePair	An edge pair (a pair of two edges)
DPath	A path class
DPoint	A point class with double (floating-point) coordinates
DPolygon	A polygon class
DSimplePolygon	A simple polygon class
DText	A text object
DTrans	A simple transformation

DVector	A vector class with double (floating-point) coordinates
DeepShapeStore	An opaque layout heap for the deep region processor
Device	A device inside a circuit.
DeviceAbstract	A geometrical device abstract
DeviceAbstractRef	Describes an additional device abstract reference for combined devices.
DeviceClass	A class describing a specific type of device.
DeviceClassBJT3Transistor	A device class for a bipolar transistor.
DeviceClassBJT4Transistor	A device class for a 4-terminal bipolar transistor.
DeviceClassCapacitor	A device class for a capacitor.
DeviceClassCapacitorWithBulk	A device class for a capacitor with a bulk terminal (substrate, well).
DeviceClassDiode	A device class for a diode.
DeviceClassFactory	A factory for creating specific device classes for the standard device extractors
DeviceClassInductor	A device class for an inductor.
DeviceClassMOS3Transistor	A device class for a 3-terminal MOS transistor.
DeviceClassMOS4Transistor	A device class for a 4-terminal MOS transistor.
DeviceClassResistor	A device class for a resistor.
DeviceClassResistorWithBulk	A device class for a resistor with a bulk terminal (substrate, well).
DeviceExtractorBJT3Transistor	A device extractor for a bipolar transistor (BJT)
DeviceExtractorBJT4Transistor	A device extractor for a four-terminal bipolar transistor (BJT)
DeviceExtractorBase	The base class for all device extractors.
DeviceExtractorCapacitor	A device extractor for a two-terminal capacitor
DeviceExtractorCapacitorWithBulk	A device extractor for a capacitor with a bulk terminal
DeviceExtractorDiode	A device extractor for a planar diode
DeviceExtractorMOS3Transistor	A device extractor for a three-terminal MOS transistor
DeviceExtractorMOS4Transistor	A device extractor for a four-terminal MOS transistor
DeviceExtractorResistor	A device extractor for a two-terminal resistor
DeviceExtractorResistorWithBulk	A device extractor for a resistor with a bulk terminal
DeviceParameterDefinition	A parameter descriptor
DeviceReconnectedTerminal	Describes a terminal rerouting in combined devices.

DeviceTerminalDefinition	A terminal descriptor
Edge	An edge class
EdgeFilter	A generic edge filter adaptor
EdgeMode	This class represents the edge mode type for \Region#edges.
EdgeOperator	A generic edge-to-polygon operator
EdgePair	An edge pair (a pair of two edges)
EdgePairFilter	A generic edge pair filter adaptor
EdgePairOperator	A generic edge-pair operator
EdgePairToEdgeOperator	A generic edge-pair-to-edge operator
EdgePairToPolygonOperator	A generic edge-pair-to-polygon operator
EdgePairs	EdgePairs (a collection of edge pairs)
EdgeProcessor	The edge processor (boolean, sizing, merge)
EdgeToEdgePairOperator	A generic edge-to-edge-pair operator
EdgeToPolygonOperator	A generic edge-to-polygon operator
Edges	A collection of edges (Not necessarily describing closed contours)
Edges::EdgeType	This enum specifies the edge type for edge angle filters.
EqualDeviceParameters	A device parameter equality comparer.
GenericDeviceCombiner	A class implementing the combination of two devices (parallel or serial mode).
GenericDeviceExtractor	The basic class for implementing custom device extractors.
GenericDeviceParameterCompare	A class implementing the comparison of device parameters.
GenericNetlistCompareLogger	An event receiver for the netlist compare feature.
HAlign	This class represents the horizontal alignment modes.
ICplxTrans	A complex transformation
IMatrix2d	A 2d matrix object used mainly for representing rotation and shear transformations (integer coordinate version).
IMatrix3d	A 3d matrix object used mainly for representing rotation, shear, displacement and perspective transformations (integer coordinate version).
InstElement	An element in an instantiation path
Instance	An instance proxy
LEFDEFReaderConfiguration	Detailed LEF/DEF reader options

LayerInfo	A structure encapsulating the layer properties
LayerMap	An object representing an arbitrary mapping of physical layers to logical layers
LayerMapping	A layer mapping (source to target layout)
Layout	The layout object
LayoutDiff	The layout compare tool
LayoutMetaInfo	A piece of layout meta information
LayoutQuery	A layout query
LayoutQueryIterator	Provides the results of the query
LayoutToNetlist	A generic framework for extracting netlists from layouts
LayoutToNetlist::BuildNetHierarchyMode	This class represents the LayoutToNetlist::BuildNetHierarchyMode enum
LayoutVsSchematic	A generic framework for doing LVS (layout vs. schematic)
Library	A Library
LoadLayoutOptions	Layout reader options
LoadLayoutOptions::CellConflictResolution	This enum specifies how cell conflicts are handled if a layout read into another layout and a cell name conflict arises.
LogEntryData	A generic log entry
Manager	A transaction manager class
Matrix2d	A 2d matrix object used mainly for representing rotation and shear transformations.
Matrix3d	A 3d matrix object used mainly for representing rotation, shear, displacement and perspective transformations.
Metrics	This class represents the metrics type for \Region#width and related checks.
Net	A single net.
NetElement	A net element for the NetTracer net tracing facility
NetPinRef	A connection to an outgoing pin of the circuit.
NetSubcircuitPinRef	A connection to a pin of a subcircuit.
NetTerminalRef	A connection to a terminal of a device.
NetTracer	The net tracer feature
NetTracerConnectionInfo	Represents a single connection info line for the net tracer technology definition
NetTracerConnectivity	A connectivity description for the net tracer

NetTracerSymbolInfo	Represents a single symbol info line for the net tracer technology definition
NetTracerTechnologyComponent	Represents the technology information for the net tracer.
Netlist	The netlist top-level class
NetlistCompareLogger	A base class for netlist comparer event receivers
NetlistComparer	Compares two netlists
NetlistCrossReference	Represents the identity mapping between the objects of two netlists.
NetlistCrossReference::CircuitPairData	A circuit match entry.
NetlistCrossReference::DevicePairData	A device match entry.
NetlistCrossReference::NetPairData	A net match entry.
NetlistCrossReference::NetPinRefPair	A match entry for a net pin pair.
NetlistCrossReference::NetSubcircuitPinRefPair	A match entry for a net subcircuit pin pair.
NetlistCrossReference::NetTerminalRefPair	A match entry for a net terminal pair.
NetlistCrossReference::PinPairData	A pin match entry.
NetlistCrossReference::Status	This class represents the NetlistCrossReference::Status enum
NetlistCrossReference::SubCircuitPairData	A subcircuit match entry.
NetlistDeviceExtractorLayerDefinition	Describes a layer used in the device extraction
NetlistObject	The base class for some netlist objects.
NetlistReader	Base class for netlist readers
NetlistSpiceReader	Implements a netlist Reader for the SPICE format.
NetlistSpiceReaderDelegate	Provides a delegate for the SPICE reader for translating device statements
NetlistSpiceWriter	Implements a netlist writer for the SPICE format.
NetlistSpiceWriterDelegate	Provides a delegate for the SPICE writer for doing special formatting for devices
NetlistWriter	Base class for netlist writers
PCellDeclaration	A PCell declaration providing the parameters and code to produce the PCell
PCellDeclarationHelper	A helper class to simplify the declaration of a PCell (Python version)
PCellDeclarationHelper	A helper class to simplify the declaration of a PCell (Ruby version)
PCellParameterDeclaration	A PCell parameter declaration
PCellParameterState	Provides access to the attributes of a single parameter within \PCellParameterStates.

PCellParameterState::ParameterStateIcon	This enum specifies the icon shown next to the parameter in PCell parameter list.
PCellParameterStates	Provides access to the parameter states inside a 'callback' implementation of a PCell
ParentInstArray	A parent instance
ParseElementComponentsData	Supplies the return value for \NetlistSpiceReaderDelegate#parse_element_components.
ParseElementData	Supplies the return value for \NetlistSpiceReaderDelegate#parse_element.
Path	A path class
Pin	A pin of a circuit.
Point	An integer point class
Polygon	A polygon class
PolygonFilter	A generic polygon filter adaptor
PolygonOperator	A generic polygon operator
PolygonToEdgeOperator	A generic polygon-to-edge operator
PolygonToEdgePairOperator	A generic polygon-to-edge-pair operator
PreferredOrientation	This class represents the PreferredOrientation enum used within polygon decomposition
PropertyConstraint	This class represents the property constraint for boolean and check functions.
RecursiveInstanceIterator	An iterator delivering instances recursively
RecursiveShapeIterator	An iterator delivering shapes recursively
Region	A region (a potentially complex area consisting of multiple polygons)
Region::OppositeFilter	This class represents the opposite error filter mode for \Region#separation and related checks.
Region::RectFilter	This class represents the error filter mode on rectangles for \Region#separation and related checks.
SaveLayoutOptions	Options for saving layouts
Severity	This enum specifies the severity level for log entries.
Shape	An object representing a shape in the layout database
ShapeCollection	A base class for the shape collections (\Region, \Edges, \EdgePairs and \Texts)
ShapeProcessor	The shape processor (boolean, sizing, merge on shapes)

Shapes	A collection of shapes
SimplePolygon	A simple polygon class
SubCircuit	A subcircuit inside a circuit.
Technology	Represents a technology
TechnologyComponent	A part of a technology definition
Text	A text object
TextFilter	A generic text filter adaptor
TextGenerator	A text generator class
TextOperator	A generic text operator
TextToPolygonOperator	A generic text-to-polygon operator
Texts	Texts (a collection of texts)
TileOutputReceiver	A receiver abstraction for the tiling processor.
TilingProcessor	A processor for layout which distributes tasks over tiles
Trans	A simple transformation
TrapezoidDecompositionMode	This class represents the TrapezoidDecompositionMode enum used within trapezoid decomposition
Utils	This namespace provides a collection of utility functions
VAlign	This class represents the vertical alignment modes.
VCplxTrans	A complex transformation
Vector	A integer vector class
ZeroDistanceMode	This class represents the zero_distance_mode type for \Region#width and related checks.

4.244. Class Index for Module lay

KLayout classes

AbstractMenu	An abstraction for the application menus
Action	The abstraction for an action (i.e. used inside menus)
Annotation	A layout annotation (i.e. ruler)
Application	The application object
BitmapBuffer	A simplistic pixel buffer representing monochrome image
BrowserDialog	A HTML display and browser dialog
BrowserPanel	A HTML display and browser widget
BrowserSource	The BrowserDialog's source for "int" URL's
ButtonState	The namespace for the button state flags in the mouse events of the Plugin class.
CellView	A class describing what is shown inside a layout view
Cursor	The namespace for the cursor constants
D25View	The 2.5d View Dialog
Dispatcher	Root of the configuration space in the plugin context and menu dispatcher
FileDialog	Various methods to request a file name
HelpDialog	The help dialog
HelpSource	A BrowserSource implementation delivering the help text for the help dialog
Image	An image to be stored as a layout annotation
ImageDataMapping	A structure describing the data mapping of an image object
InputDialog	Various methods to open a dialog requesting data entry
KeyCode	The namespace for the some key codes.
LayerProperties	The layer properties structure
LayerPropertiesIterator	Layer properties iterator
LayerPropertiesNode	A layer properties node structure
LayerPropertiesNodeRef	A class representing a reference to a layer properties node
LayoutView	The view object presenting one or more layout objects
LayoutView::SelectionMode	Specifies how selected objects interact with already selected ones.

LayoutViewWidget	
Macro	A macro class
Macro::Format	Specifies the format of a macro
Macro::Interpreter	Specifies the interpreter used for executing a macro
MacroExecutionContext	Support for various debugger features
MacroInterpreter	A custom interpreter for a DSL (domain specific language)
MainWindow	The main application window and central controller object
Marker	The floating-point coordinate marker object
MessageBox	Various methods to display message boxes
NetlistBrowserDialog	Represents the netlist browser dialog.
NetlistObjectPath	An object describing the instantiation of a netlist object.
NetlistObjectsPath	An object describing the instantiation of a single netlist object or a pair of those.
ObjectInstPath	A class describing a selected shape or instance
PixelBuffer	A simplistic pixel buffer representing an image of ARGB32 or RGB32 values
Plugin	The plugin object
PluginFactory	The plugin framework's plugin factory object

4.245. Class Index for Module rdb

KLayout classes

RdbCategory	A category inside the report database
RdbCell	A cell inside the report database
RdbItem	An item inside the report database
RdbItemValue	A value object inside the report database
RdbReference	A cell reference inside the report database
ReportDatabase	The report database object

4.246. Class Index for Module tl

KLayout classes

AbsoluteProgress	A progress reporter counting progress in absolute units
AbstractProgress	The abstract progress reporter
EmptyClass	
Executable	A generic executable object
Expression	Evaluation of Expressions
ExpressionContext	Represents the context of an expression evaluation
GlobPattern	A glob pattern matcher
Interpreter	A generalization of script interpreters
Logger	A logger
Progress	A progress reporter
Recipe	A facility for providing reproducible recipes
RelativeProgress	A progress reporter counting progress in relative units
Timer	A timer (stop watch)
Value	Encapsulates a value (preferably a plain data type) in an object