Klayout DRC - Flat File Documentation

Design Rule Checks (DRC) Basics

- Basic scripts
- Installing and running scripts

- Using KLayout as a standalone DRC engine

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.
- Cell filtering, local (region-constrained) operation.
- A tiling approach for large layouts which can be configured to make use of multiple CPU cores.

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.

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.

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:

```
# 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.

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, output, report, target
- DRC functions:
  width, space, separation (sep), notch, isolated (iso), enclosure (enc), overlap
- Boolean operations:
  & (and), | (or), - (not), ^ (xor), + (xor), join
- Sizing:
  size, sized
- Merging:
  merge, merged
- Shape selections:
  in, inside, interacting, outside, touching, overlapping
  These methods are available as in-place operations as well:
  select_interacting, select_inside, select_outside, select_touching, 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_perimeterwithout_angle, without_length
- 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

Edge and polygon layers

KLayout knows two basic layer types: polygon and edge layers. Input from layout is always of polygon type initially.

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.

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.

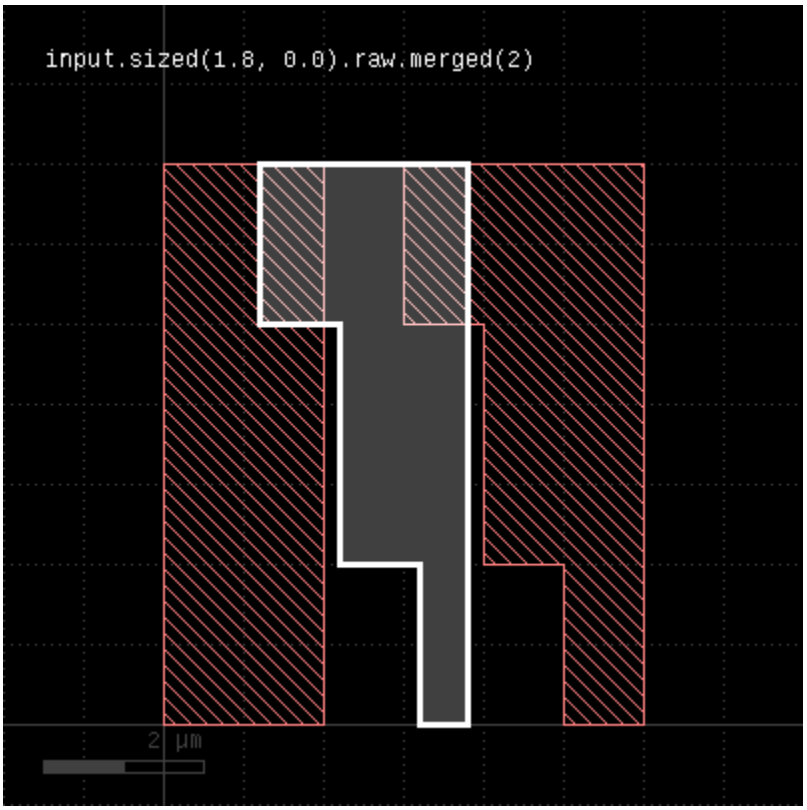
Clean mode is the default - every layer generated or taken from input will be used in clean mode. To switch to raw mode, use the "raw" method. "raw mode" is basically a flag set on the layer object which instructs the engine not the merge polygons prior to use. The raw mode flag can be reset with the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



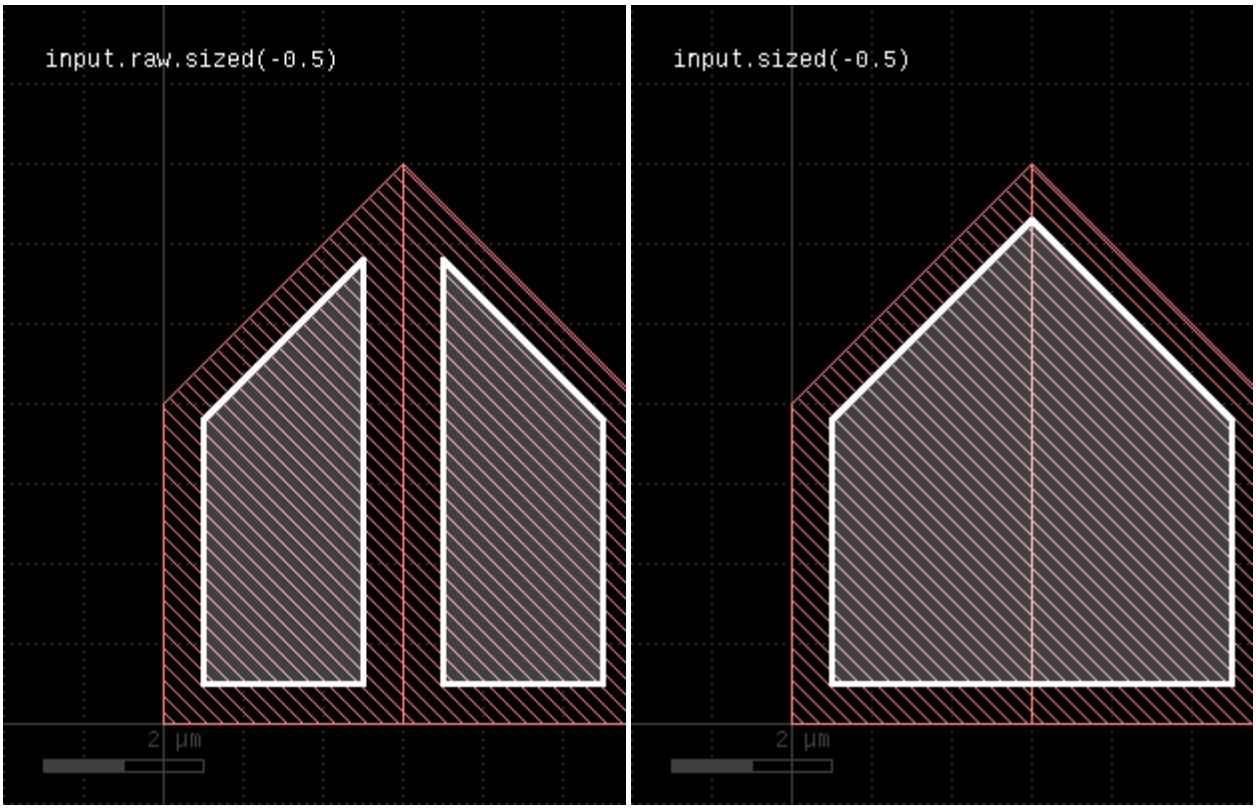
The "merged" method with an argument of 2 will produce output where more than two polygons overlap. The size function by default creates a clean layer, but separate polygons for each input polygon, so by using "raw", the layer is switched into raw mode that makes the individual polygons accessible without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



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 enterin 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.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 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.

DRC Reference

DRC Reference: Layer Object

The layer object represents a collection of polygons, edges or edge pairs.

- "extended" - Returns polygons describing an area along the edges of the input
- "extended_in" - Returns polygons describing an area along the edges of the input
- "extended_out" - Returns polygons describing an area along the edges of the input
- "extent_refs" - Returns partial references to the boundings boxes of the polygons
- "extents" - Returns the bounding box of each input object
- "first_edges" - Returns the first edges of an edge pair collection
- "flatten" - Flattens the layer
- "holes" - Selects all polygon holes from the input
- "hulls" - Selects all polygon hulls from the input
- "in" - Selects shapes or regions of self which are contained in the other layer
- "insert" - Inserts one or many objects into the layer
- "inside" - Selects shapes or regions of self which are inside the other region
- "inside_part" - Returns the parts of the edges inside the given region
- "interacting" - Selects shapes or regions of self which touch or overlap shapes from the other region
- "is_box?" - Returns true, if the region contains a single box
- "is_clean?" - Returns true, if the layer is clean state
- "is_deep?" - Returns true, if the layer is a deep (hierarchical) layer
- "is_empty?" - Returns true, if the layer is empty
- "is_merged?" - Returns true, if the polygons of the layer are merged
- "is_raw?" - Returns true, if the layer is raw state
- "iso" - An alias for "isolated"
- "isolated" - An isolation check
- "join" - Joins the layer with another layer
- "length" - Returns the total length of the edges in the edge layer
- "merge" - Merges the layer (modifies the layer)
- "merged" - Returns the merged layer
- "middle" - Returns the center points of the bounding boxes of the polygons
- "move" - Moves (shifts, translates) a layer (modifies the layer)
- "moved" - Moves (shifts, translates) a layer
- "non_rectangles" - Selects all polygons from the input which are not rectangles
- "non_rectilinear" - Selects all non-rectilinear polygons from the input
- "non_strict" - Marks a layer for non-strict handling
- "not" - Boolean NOT operation
- "not_in" - Selects shapes or regions of self which are not contained in the other layer
- "not_inside" - Selects shapes or regions of self which are not inside the other region
- "not_interacting" - Selects shapes or regions of self which do not touch or overlap shapes from the other region
- "not_outside" - Selects shapes or regions of self which are not outside the other region
- "not_overlapping" - Selects shapes or regions of self which do not overlap shapes from the other region
- "notch" - An intra-region spacing check
- "odd_polygons" - Checks for odd polygons (self-overlapping, non-orientable)
- "ongrid" - Checks for on-grid vertices
- "or" - Boolean OR operation
- "output" - Outputs the content of the layer

- "outside" - Selects shapes or regions of self which are outside the other region
- "outside_part" - Returns the parts of the edges outside the given region
- "overlap" - An overlap check
- "overlapping" - Selects shapes or regions of self which overlap shapes from the other region
- "perimeter" - Returns the total perimeter of the polygons in the region
- "polygons" - Returns polygons from edge pairs
- "polygons?" - Returns true, if the layer is a polygon layer
- "raw" - Marks a layer as raw
- "rectangles" - Selects all rectangle polygons from the input
- "rectilinear" - Selects all rectilinear polygons from the input
- "rotate" - Rotates a layer (modifies the layer)
- "rotated" - Rotates a layer
- "rounded_corners" - Applies corner rounding to each corner of the polygon
- "scale" - Scales a layer (modifies the layer)
- "scaled" - Scales a layer
- "second_edges" - Returns the second edges of an edge pair collection
- "select" - Selects edges, edge pairs or polygons based on evaluation of a block
- "select_inside" - Selects shapes or regions of self which are inside the other region
- "select_interacting" - Selects shapes or regions of self which touch or overlap shapes from the other region
- "select_not_inside" - Selects shapes or regions of self which are not inside the other region
- "select_not_interacting" - Selects shapes or regions of self which do not touch or overlap shapes from the other region
- "select_not_outside" - Selects shapes or regions of self which are not outside the other region
- "select_not_overlapping" - Selects shapes or regions of self which do not overlap shapes from the other region
- "select_outside" - Selects shapes or regions of self which are outside the other region
- "select_overlapping" - Selects shapes or regions of self which overlap shapes from the other region
- "sep" - An alias for "separation"
- "separation" - A two-layer spacing check
- "size" - Polygon sizing (per-edge biasing, modifies the layer)
- "sized" - Polygon sizing (per-edge biasing)
- "smoothed" - Smoothes the polygons of the region
- "snap" - Brings each vertex on the given grid (g or gx/gy for x or y direction)
- "snapped" - Returns a snapped version of the layer
- "space" - A space check
- "start_segments" - Returns the part at the beginning of each edge
- "strict" - Marks a layer for strict handling
- "strict?" - Returns true, if strict handling is enabled for this layer
- "texts" - Selects texts from an original layer
- "transform" - Transforms a layer (modifies the layer)
- "transformed" - Transforms a layer

- "width" - A width check
- "with_angle" - Selects edges by their angle
- "with_area" - Selects polygons by area
- "with_bbox_height" - Selects polygons by the height of the bounding box
- "with_bbox_max" - Selects polygons by the maximum dimension of the bounding box
- "with_bbox_min" - Selects polygons by the minimum dimension of the bounding box
- "with_bbox_width" - Selects polygons by the width of the bounding box
- "with_length" - Selects edges by their length
- "with_perimeter" - Selects polygons by perimeter
- "without_angle" - Selects edges by the their angle
- "without_area" - Selects polygons by area
- "without_bbox_height" - Selects polygons by the height of the bounding box
- "without_bbox_max" - Selects polygons by the maximum dimension of the bounding box
- "without_bbox_min" - Selects polygons by the minimum dimension of the bounding box
- "without_bbox_width" - Selects polygons by the width of the bounding box
- "without_length" - Selects edges by the their length
- "without_perimeter" - Selects polygons by perimeter
- "xor" - Boolean XOR operation
- "|" - Boolean OR operation

"&" - 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

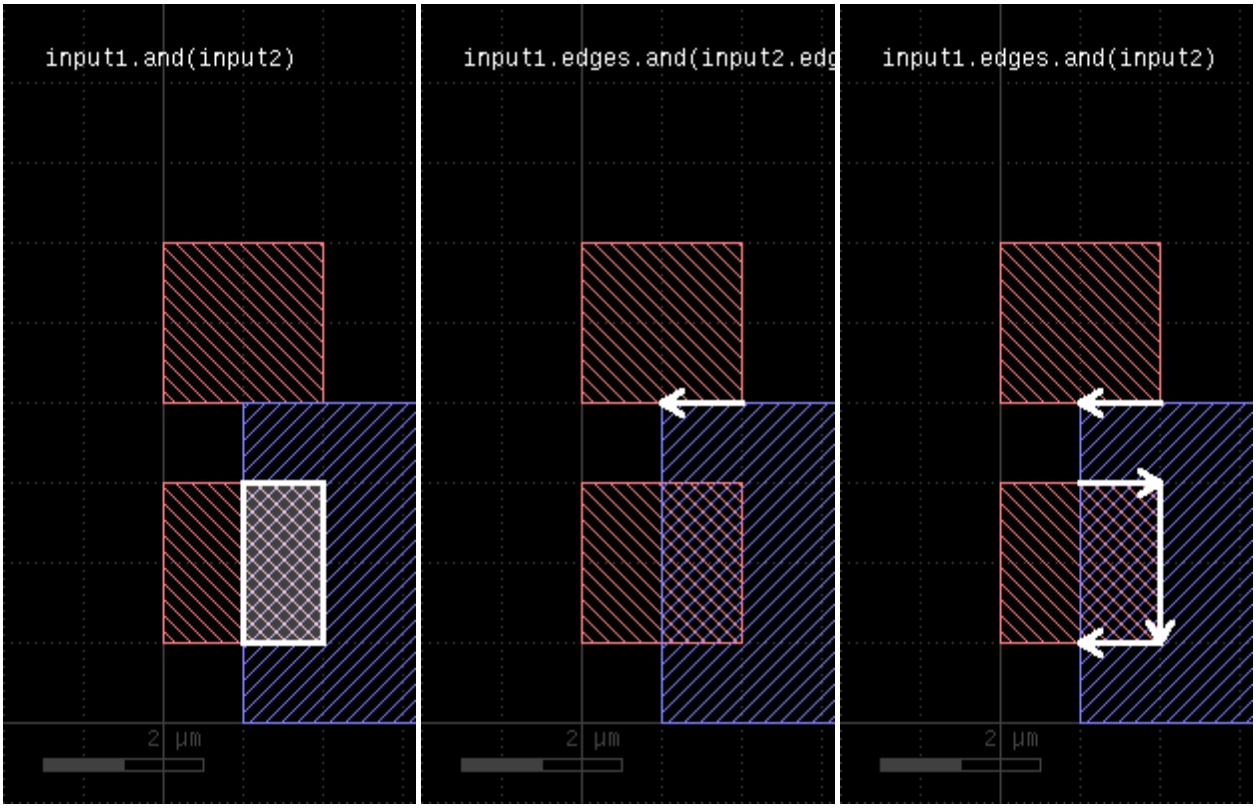
Usage:

- `layer.and(other)`

The method computes a boolean AND 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 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 (layer1: red, layer2: blue):



"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.

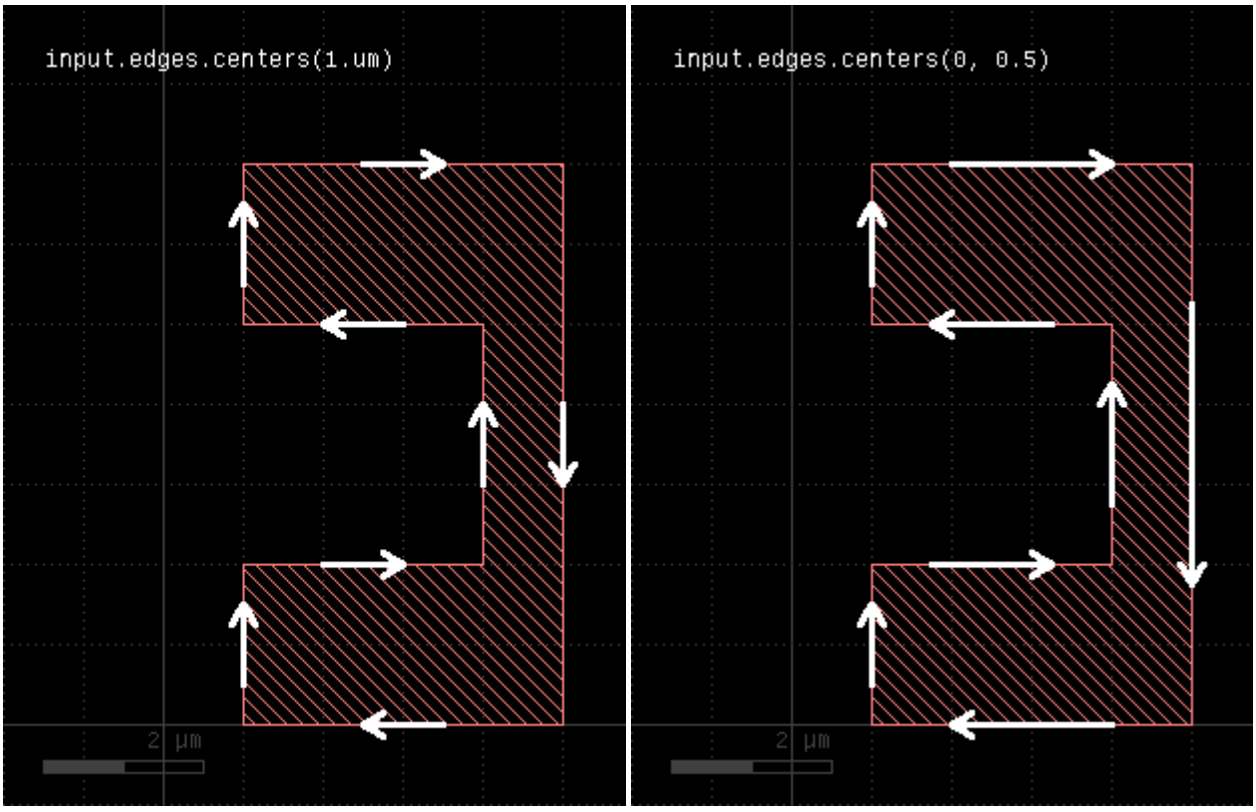
"centers" - Returns the center parts of the edges

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_polygons, 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 { |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 { |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 { |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 ])`

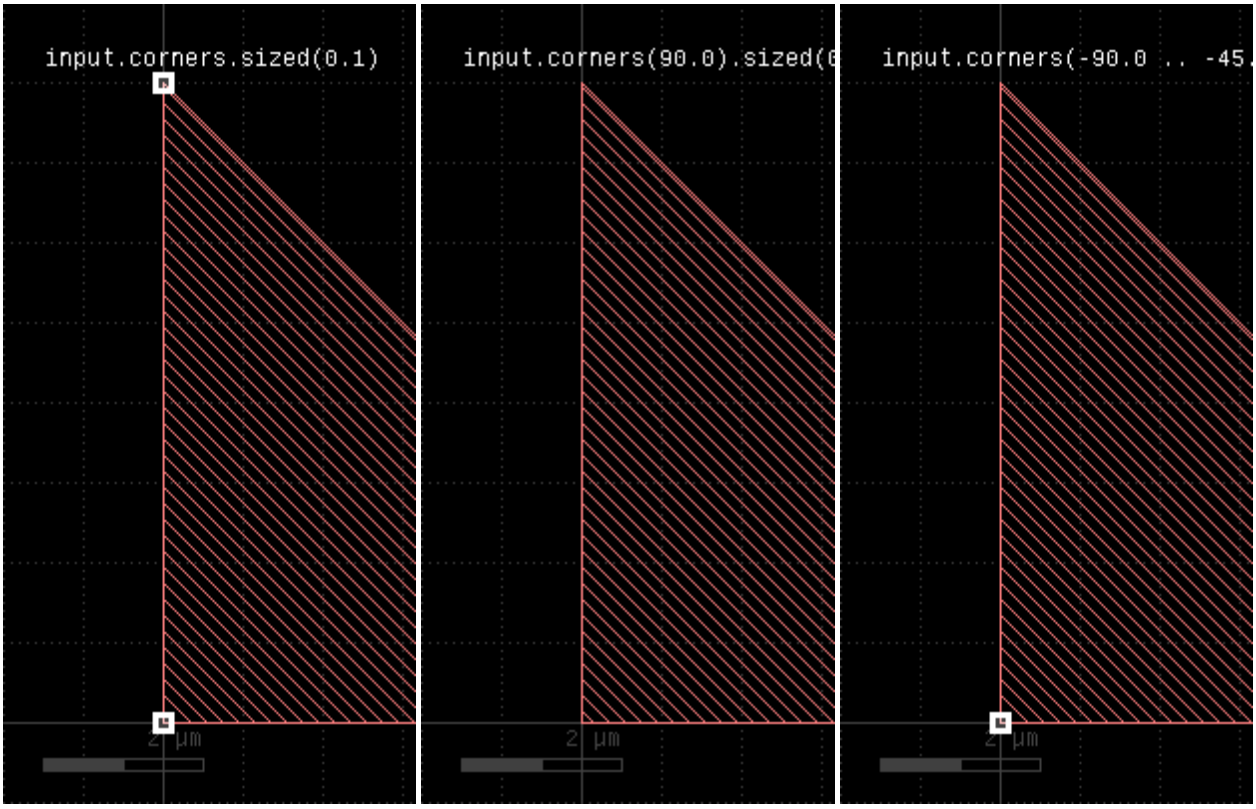
This method produces markers on the corners of the polygons. An angle criterion can be given which selects corners based on the angle of the connecting edges. Positive angles indicate a left turn while negative angles indicate a right turn. Since polygons are oriented clockwise, positive angles indicate concave corners while negative ones indicate convex 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

The following images show the effect of this method:



"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.

"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

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.

"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

"enclosing" - An enclosing check

Usage:

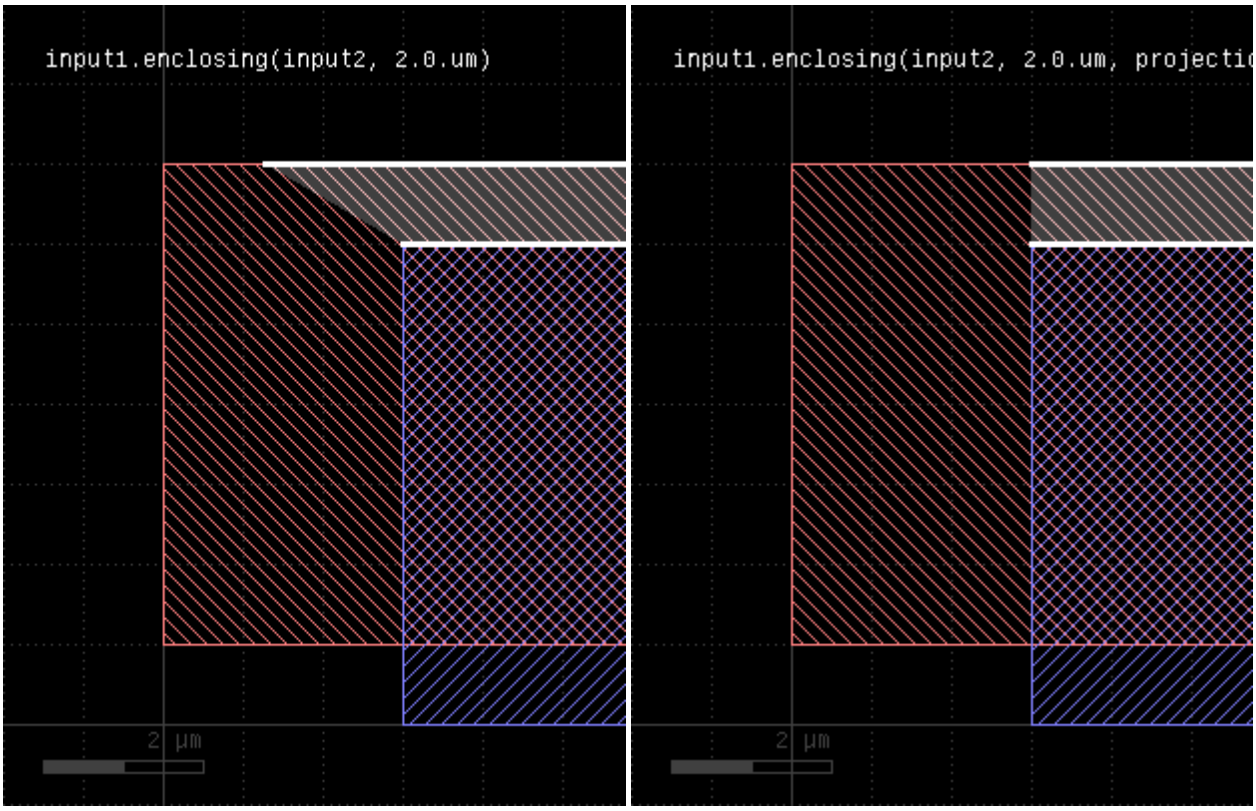
- `layer.enclosing(other_layer, value [, options])`

This method checks whether layer encloses (is bigger than) other_layer by the given dimension. Locations, where this is not the case 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.

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. The options available are the same than for width. 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

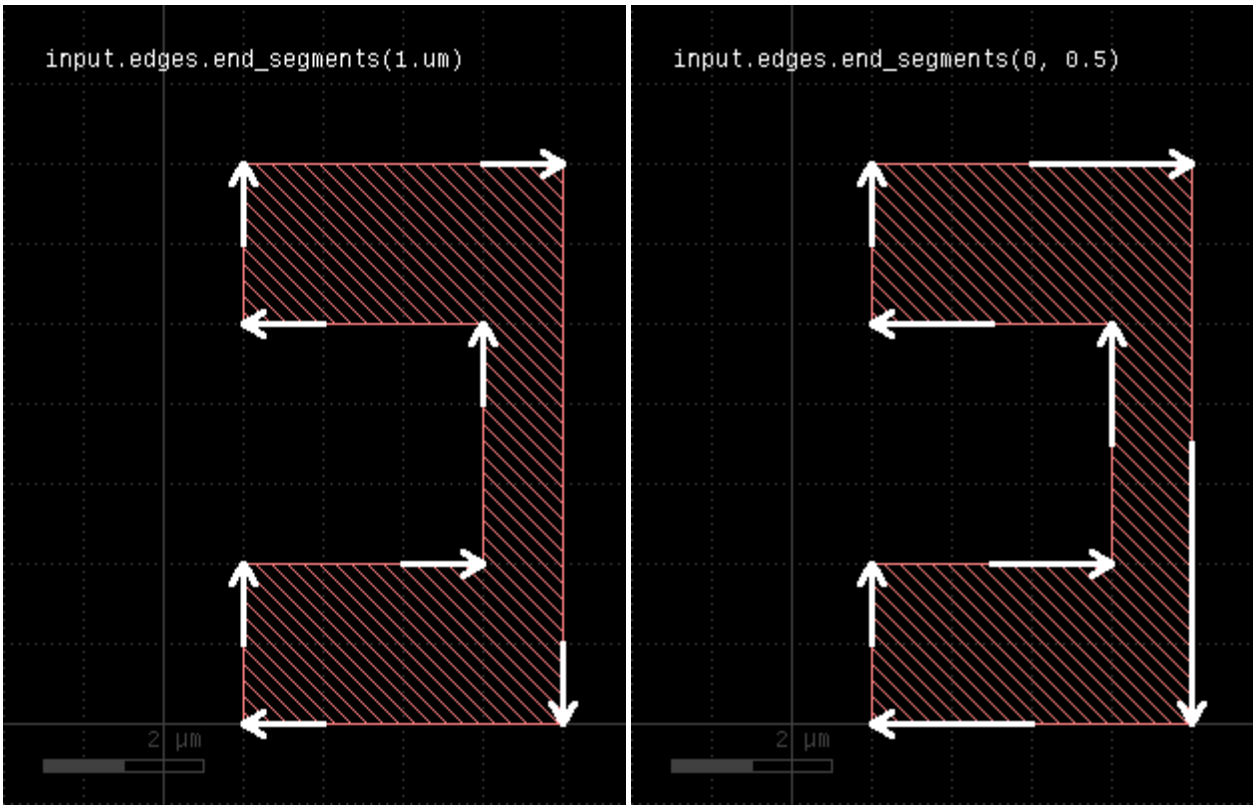
Usage:

- `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:

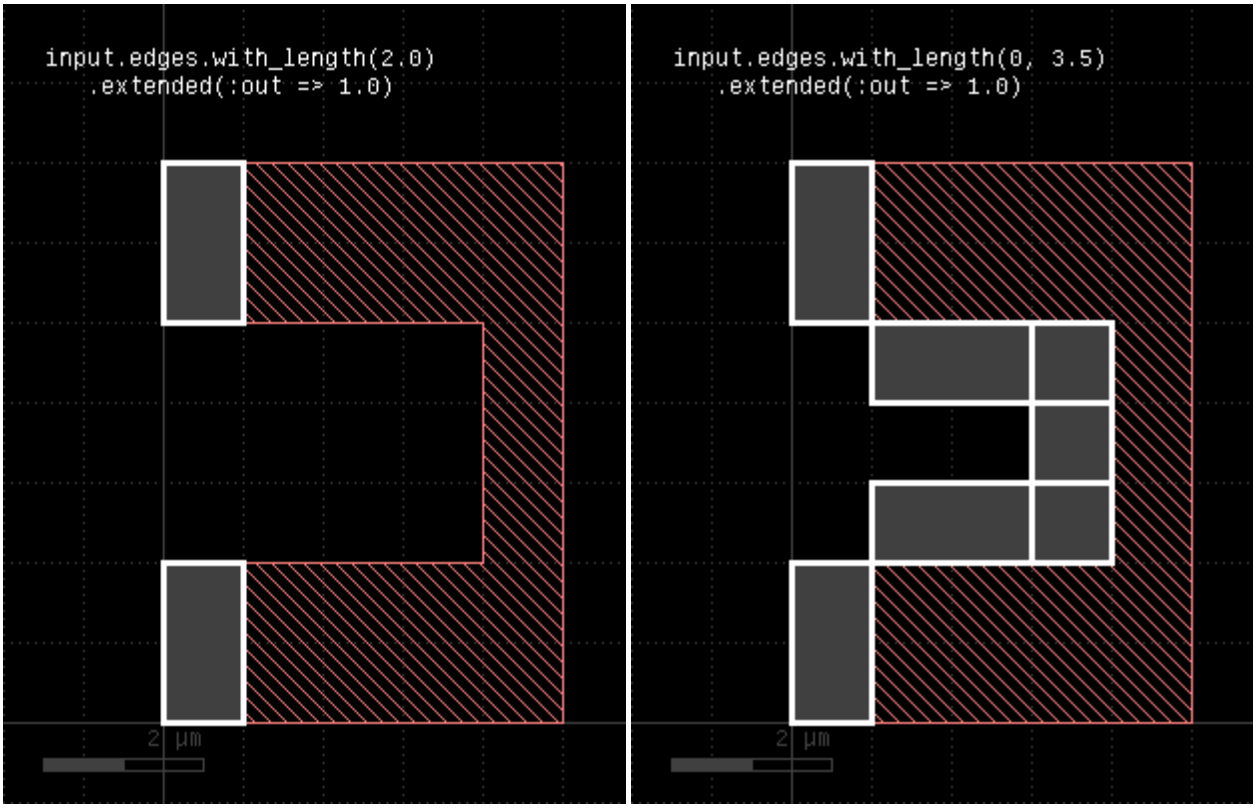
- `layer.extended([:begin => b,] [:end => e,] [:out => o,] [:in => i], [:joined => true])`
- `layer.extended(b, e, o, i)`
- `layer.extended(b, e, o, i, joined)`

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:


input.edges.with_length(2.0)
.extended(:out => 1.0)
2 µm
input.edges.with_length(0, 3.5)
.extended(:out => 1.0)
2 µm
input.edges.with_length(0, 3.5)
.extended(:out => 1.0, :joined => true)
2 µm
input.edges.with_length(2.0)
.extended(0.0, -0.5, 1.0, -0.5)
2 µm

"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

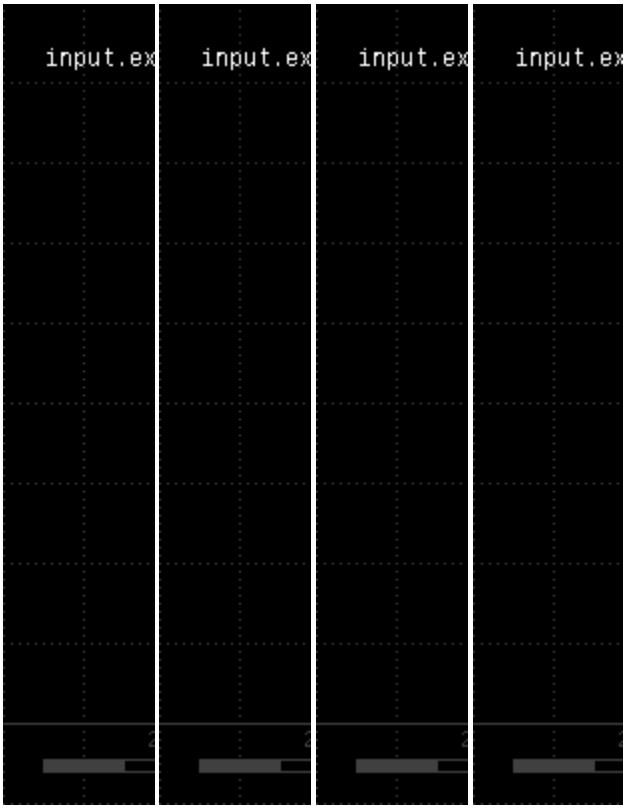
Dots are represented by small (2x2 DBU) boxes or point-like edges with edge output. Lines are represented by narrow or flat (2 DBU) boxes or edges for edge output. Edges will follow the orientation convention for the corresponding edges - i.e. "inside" of the bounding box is on the right side of the edge.

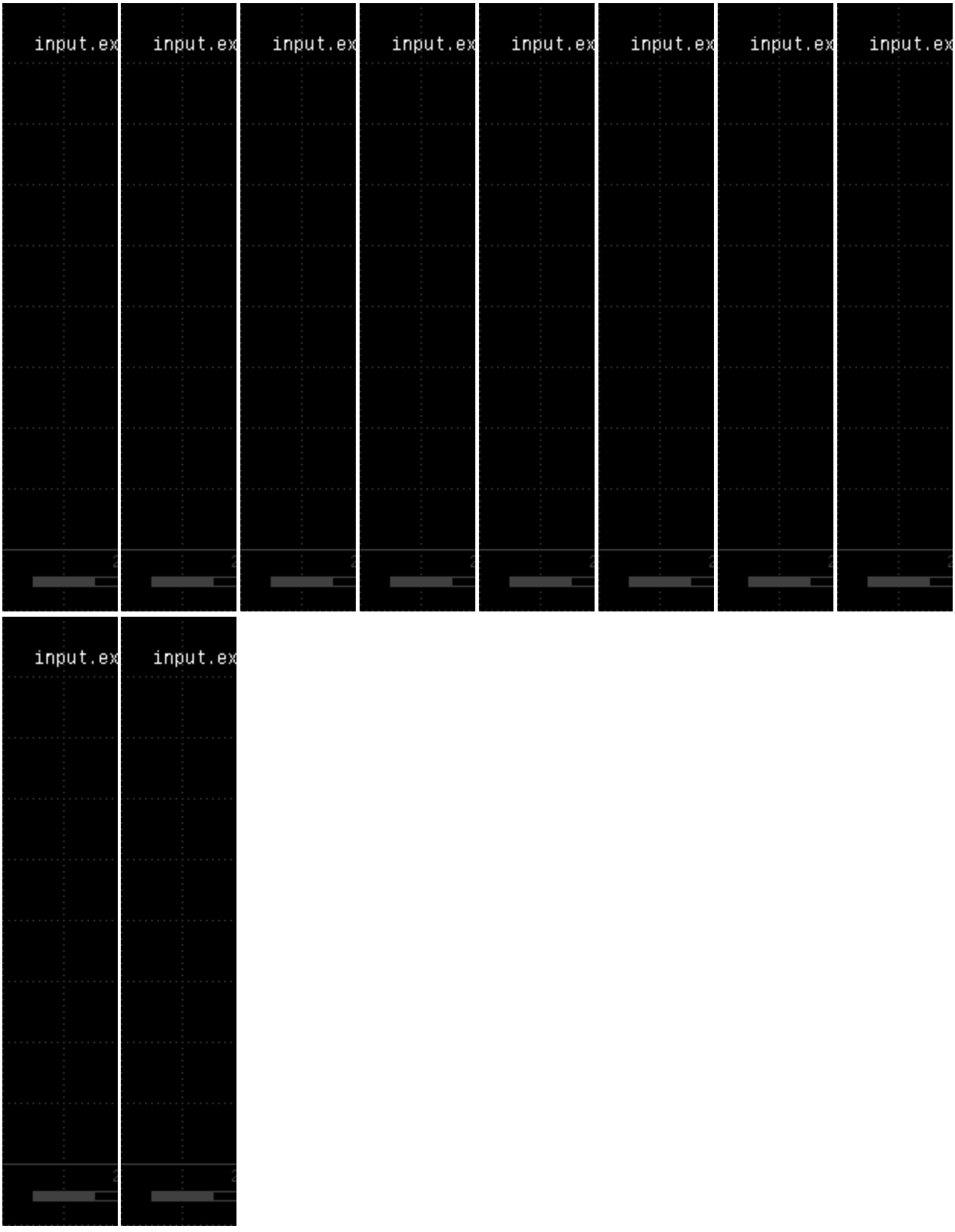
The following additional option controls the output format:

- **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:


input.ex


input.ex
input.ex
input.ex
input.ex



"extents" - Returns the bounding box of each input object

Usage:

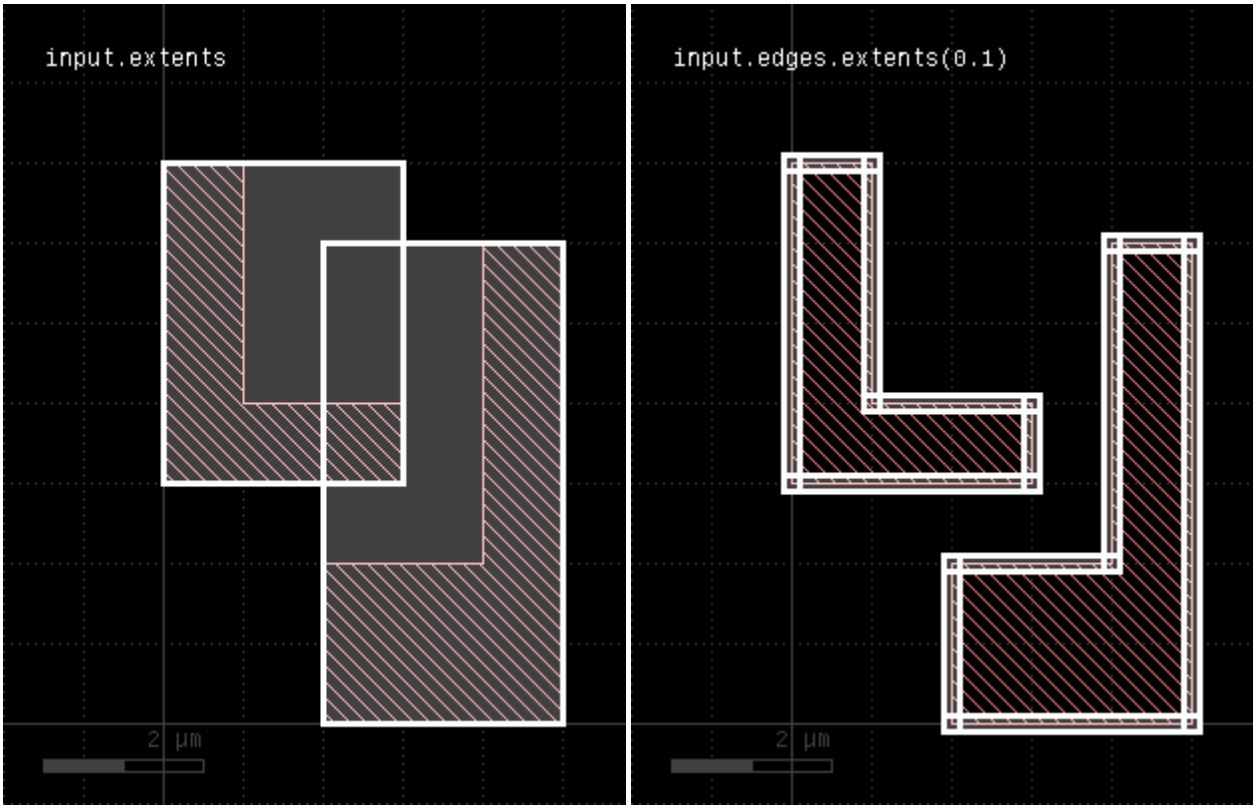
- `layer.extents([ enlargement ])`

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:



"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.

"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 polygons, edges or edge pairs.

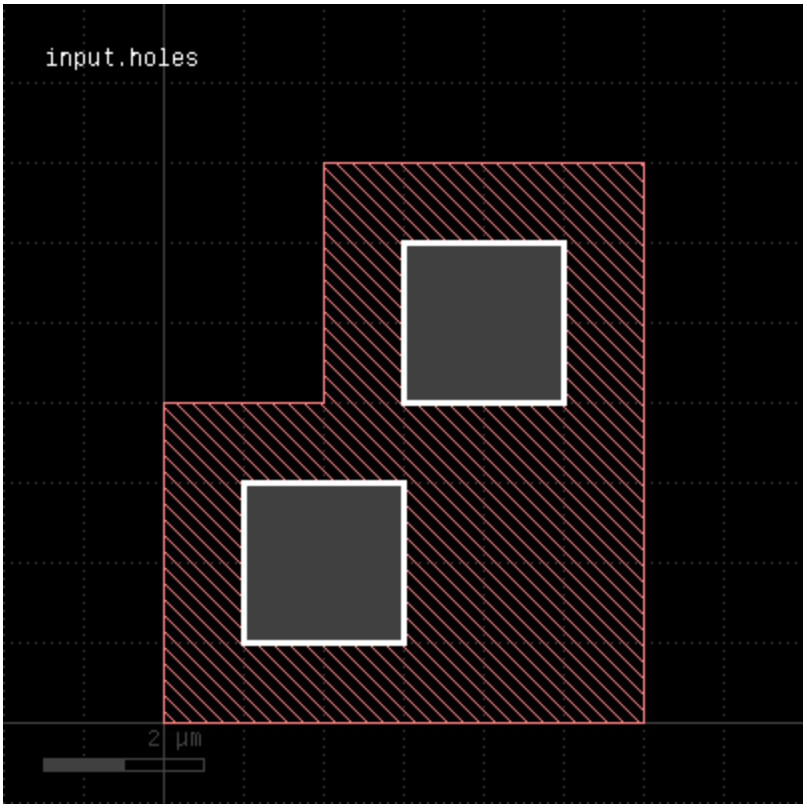
"holes" - Selects all polygon holes from the input

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:



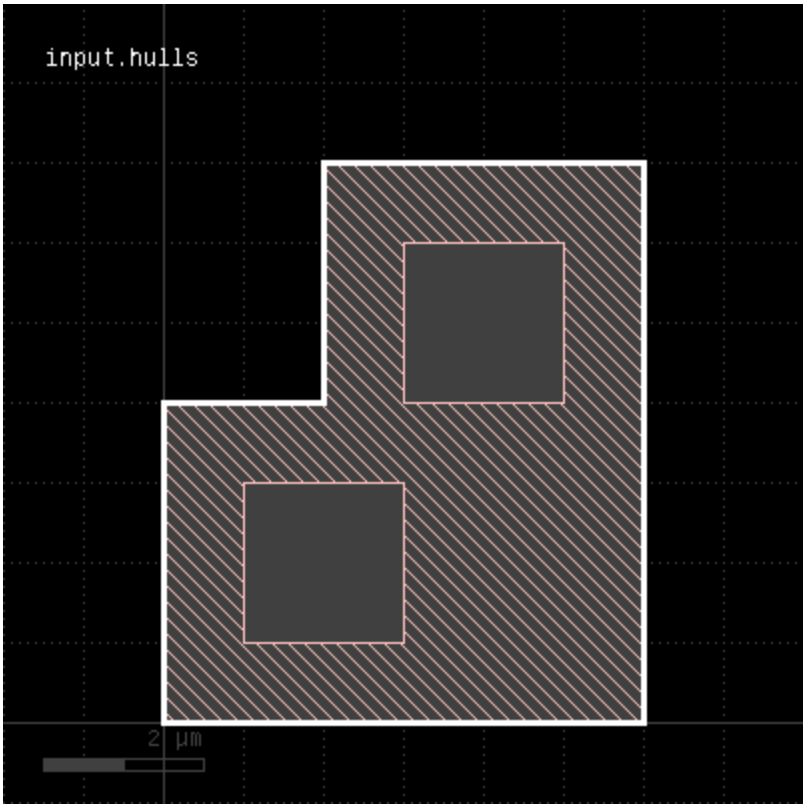
"hulls" - Selects all polygon hulls from the input

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:

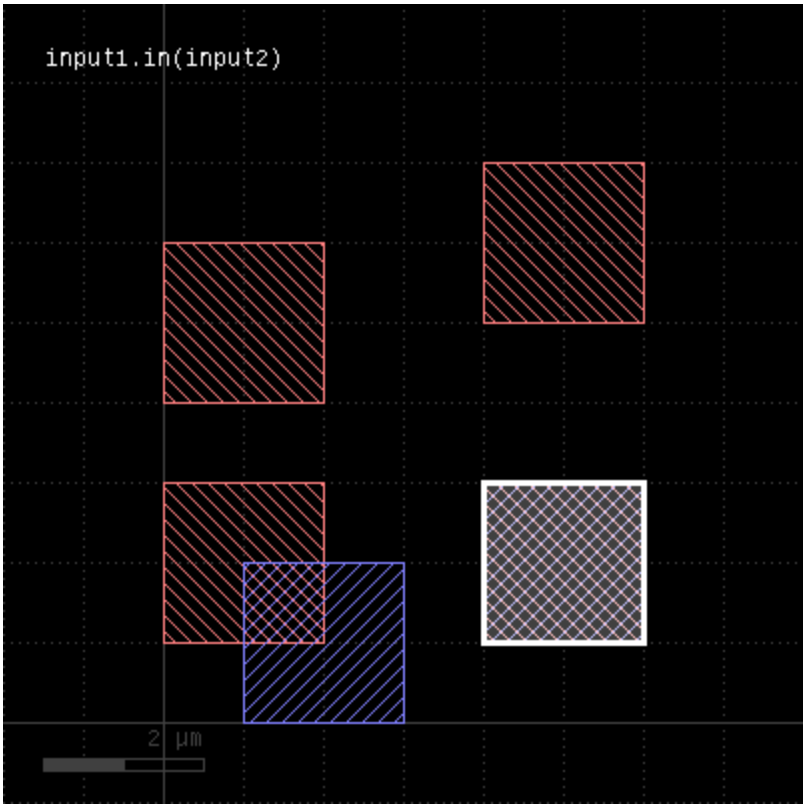
- `layer.in(other)`

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):



"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 (global#edge, global#polygon, global#box and #global#path). However, RBA constructors can used as well.

The insert method is useful in combination with the global#polygon_layer or global#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 shapes or regions of self which are inside the other region

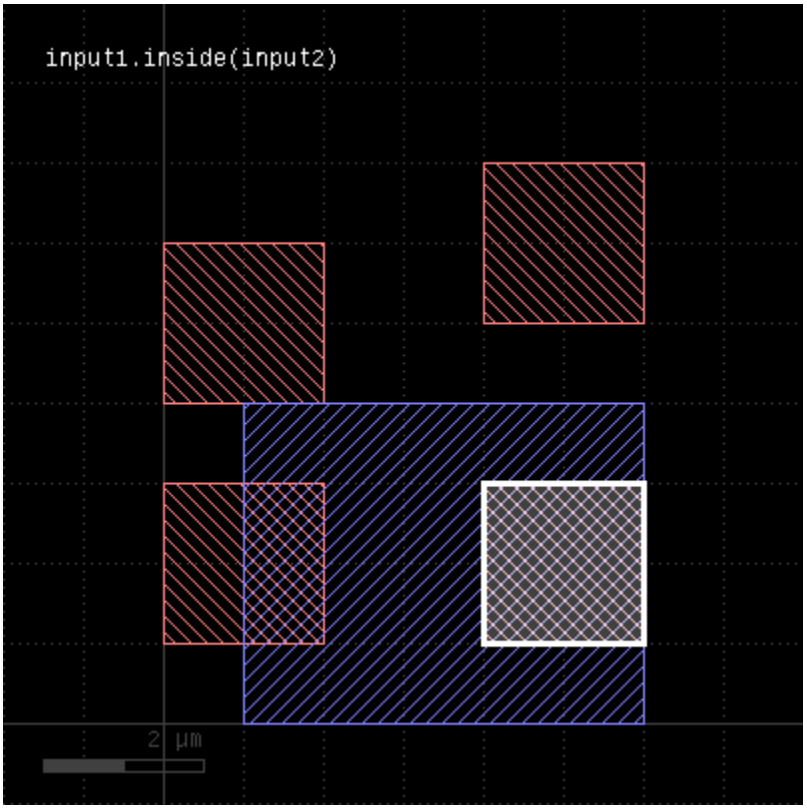
Usage:

- `layer.inside(other)`

This method selects all shapes or regions from self which are inside the other region. completely (completely covered by polygons from the other region). If self is in raw mode, this method will select individual shapes. Otherwise, this method will select coherent regions and no part of these regions may be outside the other region. It returns a new layer containing the selected shapes. A version which modifies self is select_inside.

This method is available for polygon layers.

The following image shows the effect of the "inside" method (input1: red, input2: blue):



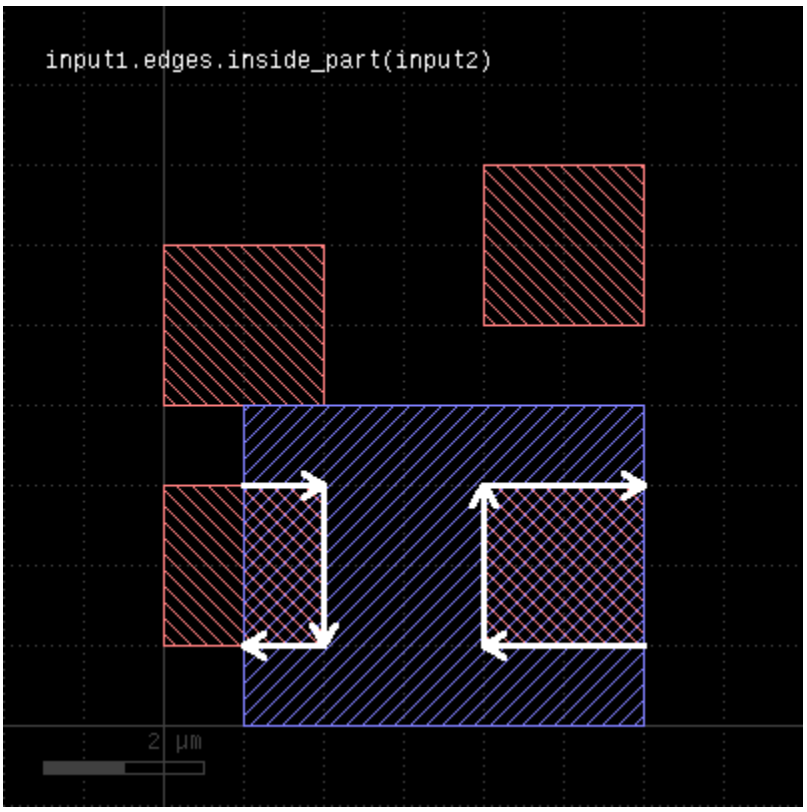
"inside_part" - Returns the parts of the edges inside the given region

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.



"interacting" - Selects shapes or regions of self which touch or overlap shapes from the other region

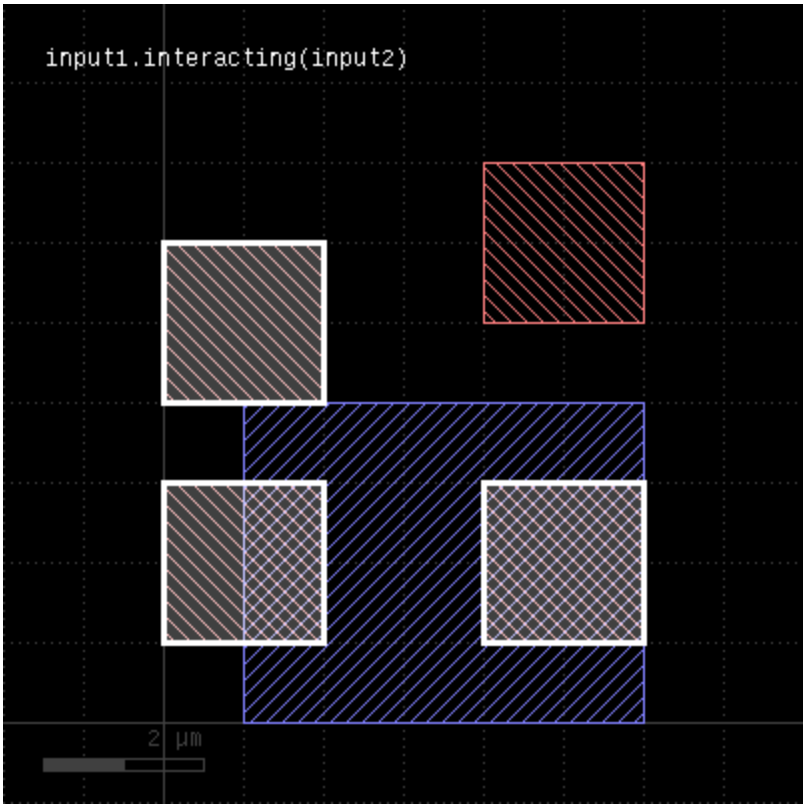
Usage:

- `layer.interacting(other)`

This method selects all shapes or regions from self which touch or overlap shapes from the other region. If 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 and edge layers. Edges can be selected with respect to other edges or polygons.

The following image shows the effect of the "interacting" method (input1: red, input2: blue):



"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

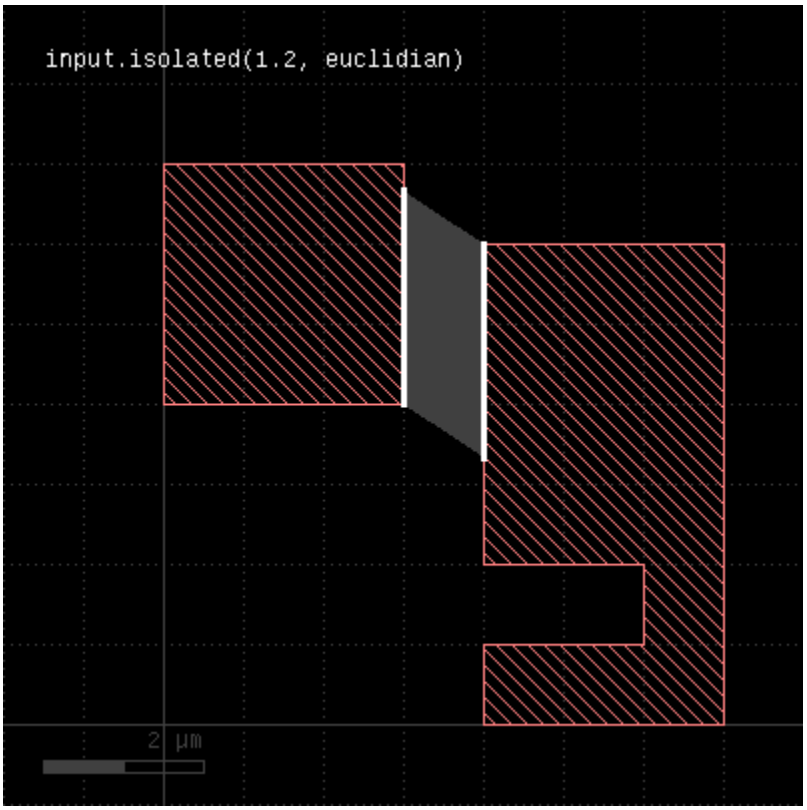
"isolated" - An isolation check

Usage:

- `layer.isolated(value [, options])`

See space for a description of this method. In contrast to space, this method is available for polygon layers only, since only on such layers different polygons can be identified.

The following image shows the effect of the isolated check:



"join" - Joins the layer with another layer

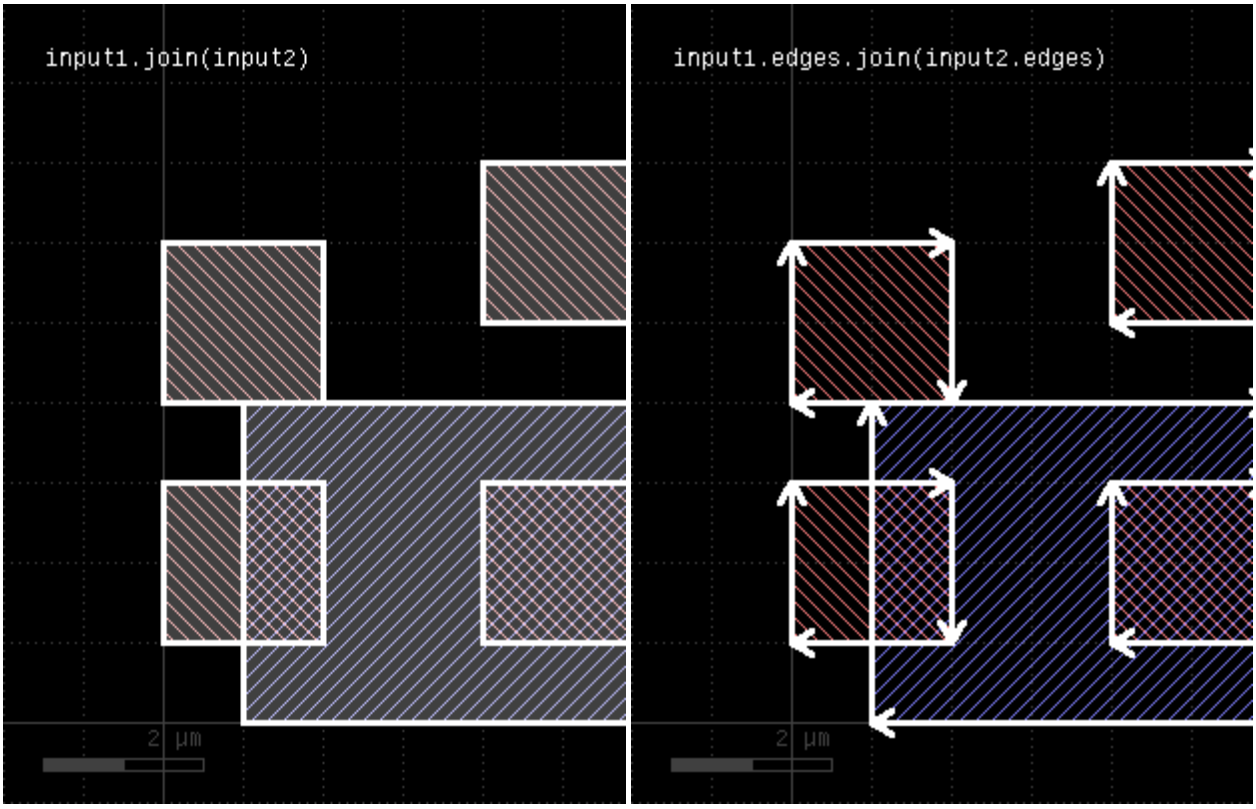
Usage:

- `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 (layer1: red, layer2: 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.

"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:

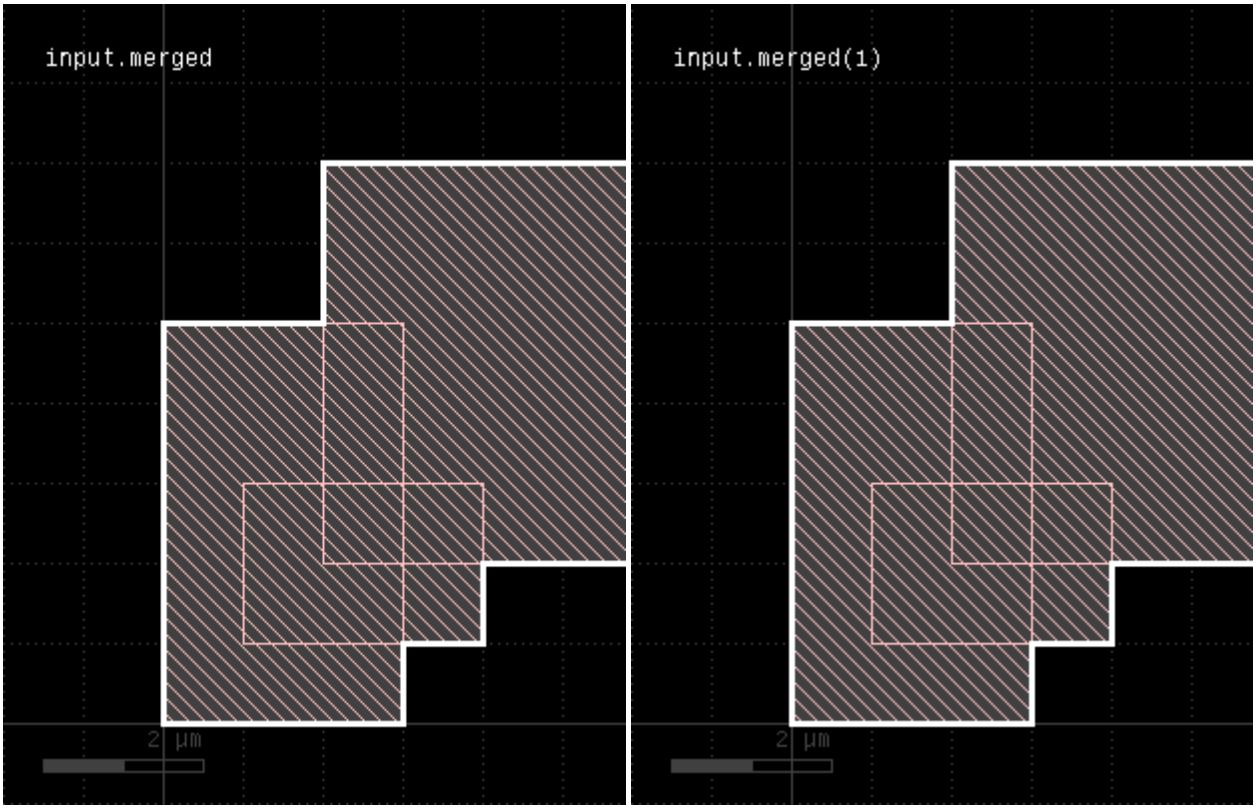
- `layer.merged([overlap_count])`

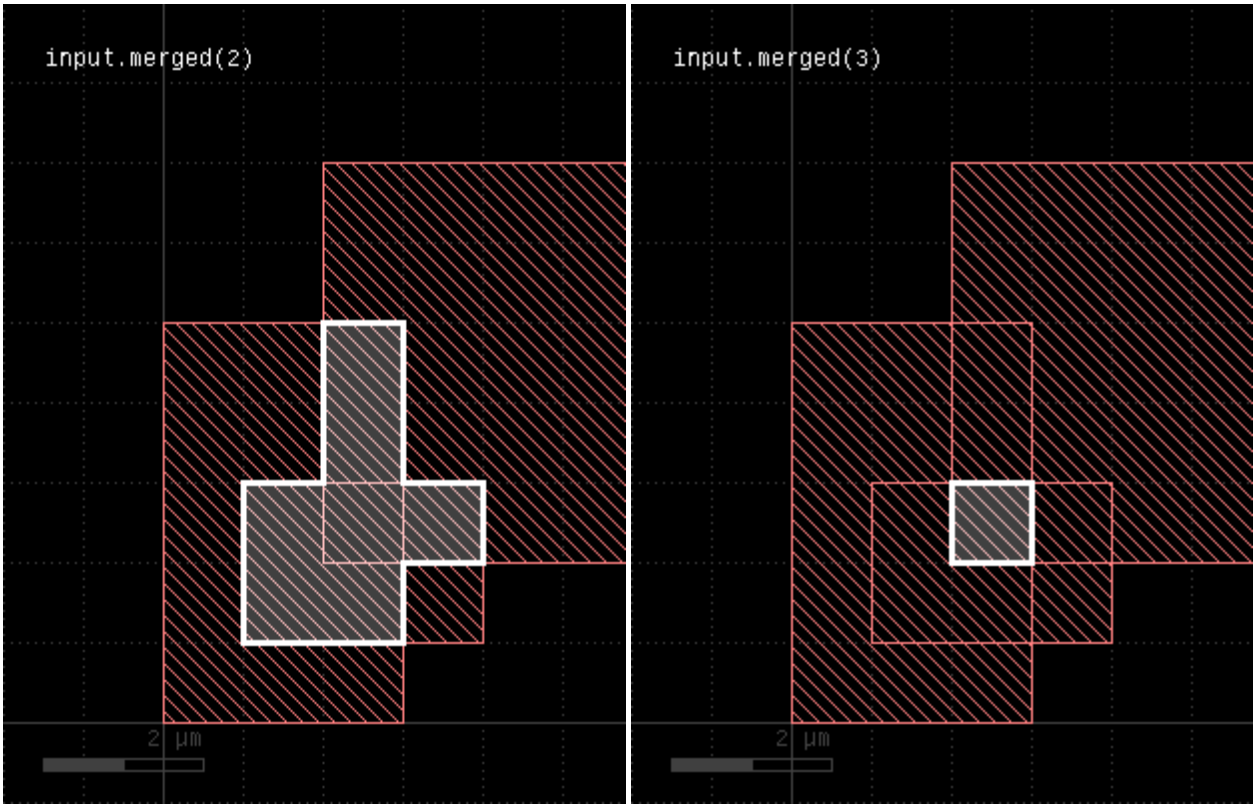
Returns the merged input. Usually, merging is done implicitly using the clean state (which is default). However, in raw state, merging can be enforced by using this method. In addition, this method allows specification of a minimum overlap count, i.e. only where at least the given number of polygons 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:

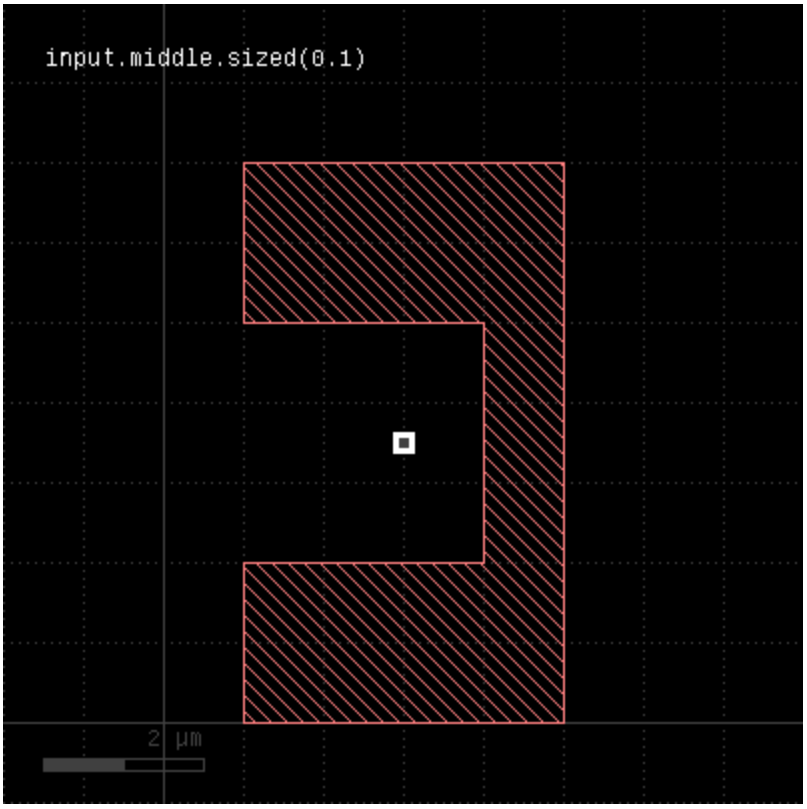
- `layer.middle([ options ])`

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

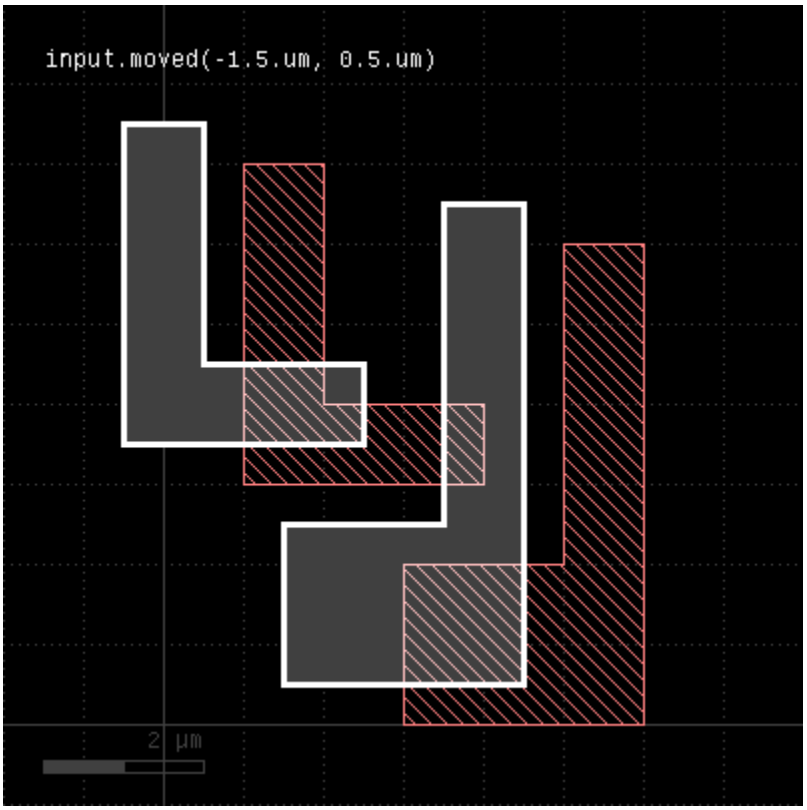
Usage:

- `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:



"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_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

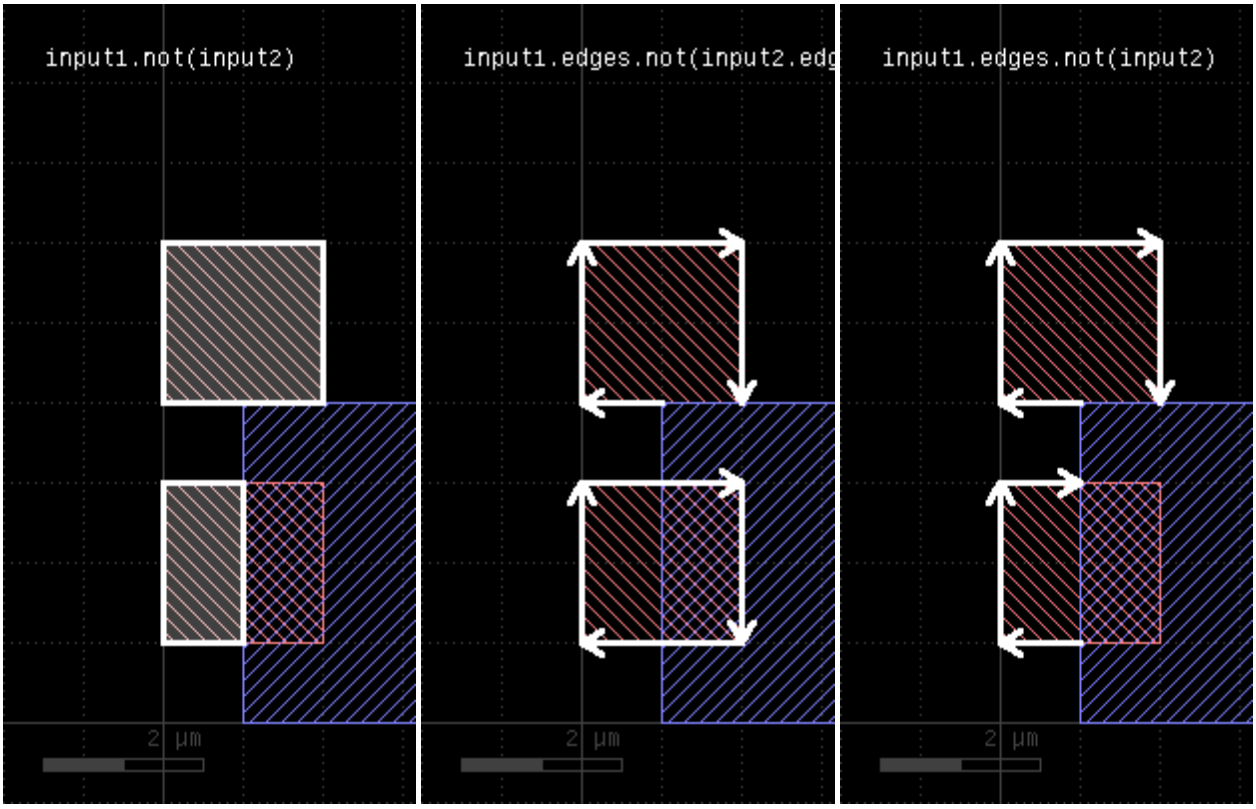
Usage:

- `layer.not(other)`

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 (layer1: red, layer2: blue):



"not_in" - Selects shapes or regions of self which are not contained in the other layer

Usage:

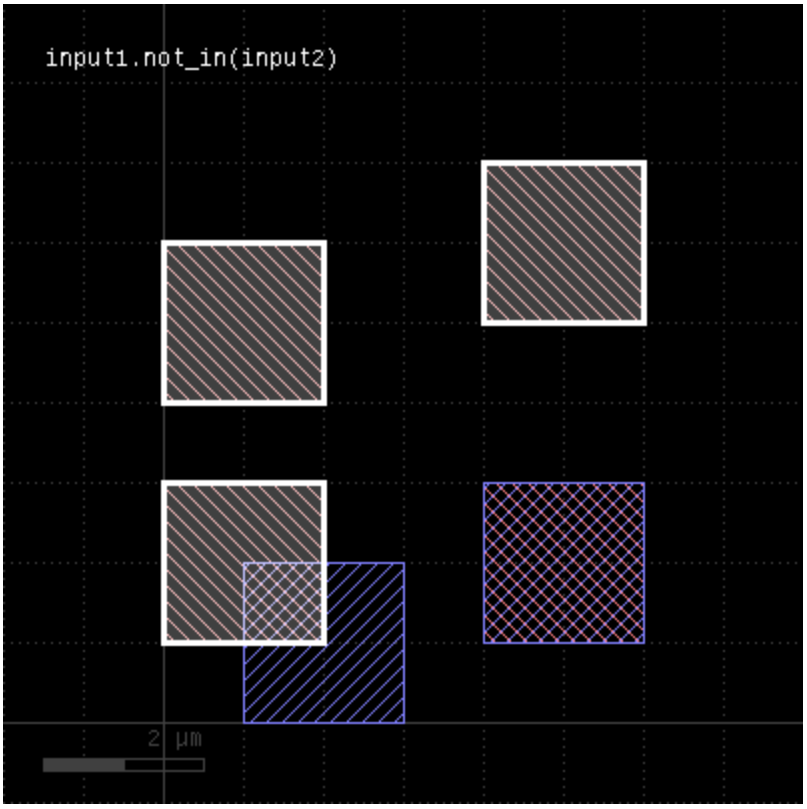
- `layer.not_in(other)`

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 shapes or regions of self which are not inside the other region

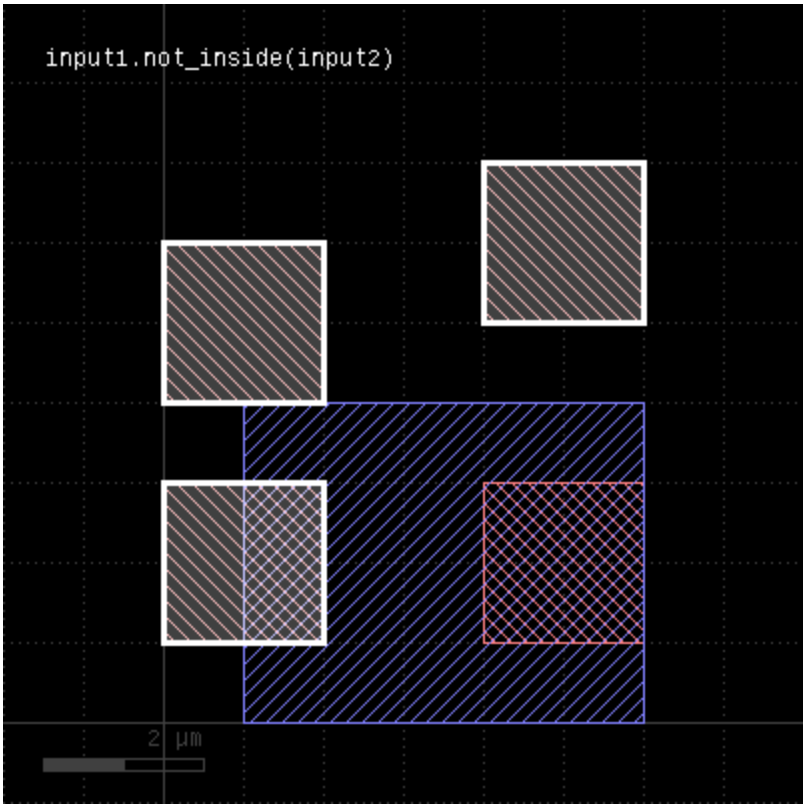
Usage:

- `layer.not_inside(other)`

This method selects all shapes or regions from self which are not inside the other region. completely (completely covered by polygons from the other region). If self is in raw mode, this method will select individual shapes. Otherwise, this method will select coherent regions and no part of these regions may be outside the other region. It returns a new layer containing the selected shapes. A version which modifies self is select_not_inside.

This method is available for polygon layers.

The following image shows the effect of the "not_inside" method (input1: red, input2: blue):



"not_interacting" - Selects shapes or regions of self which do not touch or overlap shapes from the other region

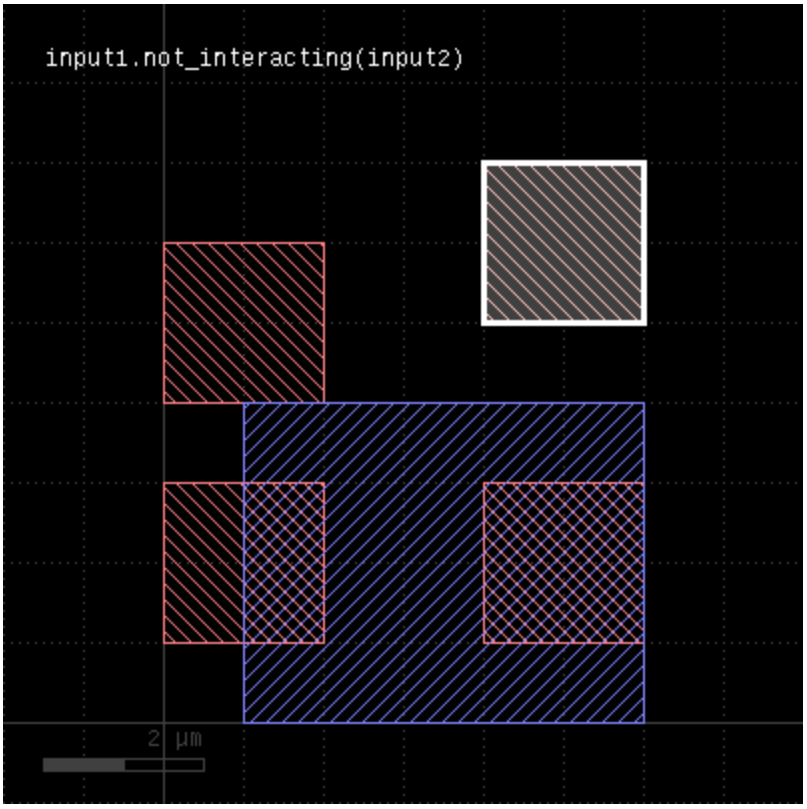
Usage:

- `layer.not_interacting(other)`

This method selects all shapes or regions from self which do not touch or overlap shapes from the other region. If 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 and edge layers. Edges can be selected with respect to other edges or polygons.

The following image shows the effect of the "not_interacting" method (input1: red, input2: blue):



"not_outside" - Selects shapes or regions of self which are not outside the other region

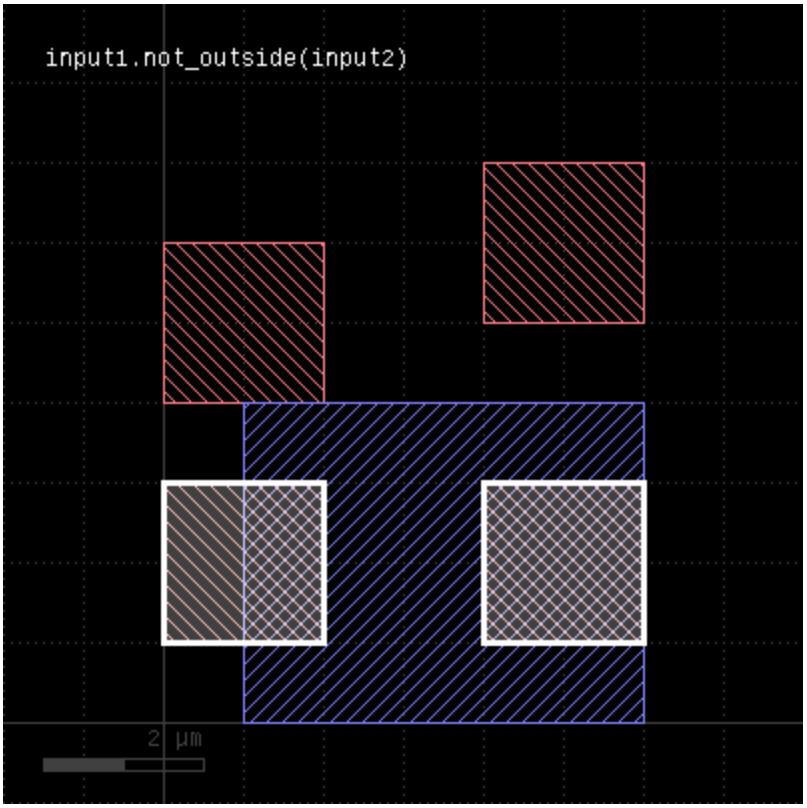
Usage:

- `layer.not_outside(other)`

This method selects all shapes or regions from self which are not completely outside the other region (part of these shapes or regions may be covered by shapes from the other region). If self is in raw mode, this method will select individual shapes. Otherwise, this method will select coherent regions and no part of these regions may overlap with shapes from the other region. It returns a new layer containing the selected shapes. A version which modifies self is select_not_outside.

This method is available for polygon layers.

The following image shows the effect of the "not_outside" method (input1: red, input2: blue):



"not_overlapping" - Selects shapes or regions of self which do not overlap shapes from the other region

Usage:

- `layer.not_overlapping(other)`

This method selects all shapes or regions from self which do not overlap shapes from the other region. If self is in raw mode (see raw), coherent regions are selected from self, otherwise individual shapes are selected.

The "not_overlapping" method is equivalent to the outside method. It is provided as an alias for consistency.

This method is available for polygon and edge layers. Edges can be selected with respect to other edges or polygons. It returns a new layer containing the selected shapes. A version which modifies self is select_not_overlapping.

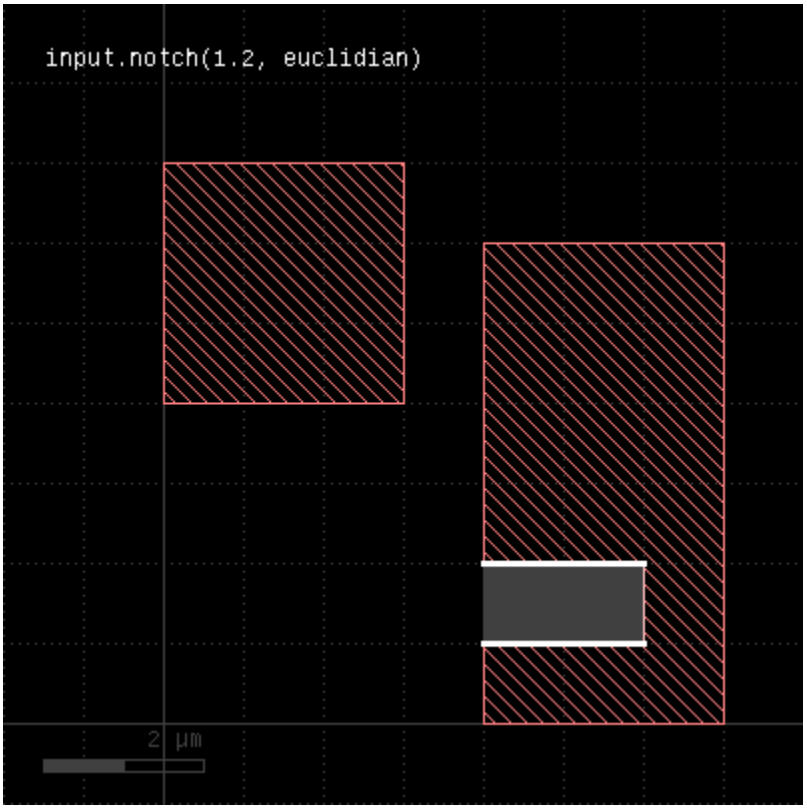
"notch" - An intra-region spacing check

Usage:

- `layer.notch(value [, options])`

See space for a description of this method. In contrast to space, this method is available for polygon layers only, since only on such layers different polygons can be identified.

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).

"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

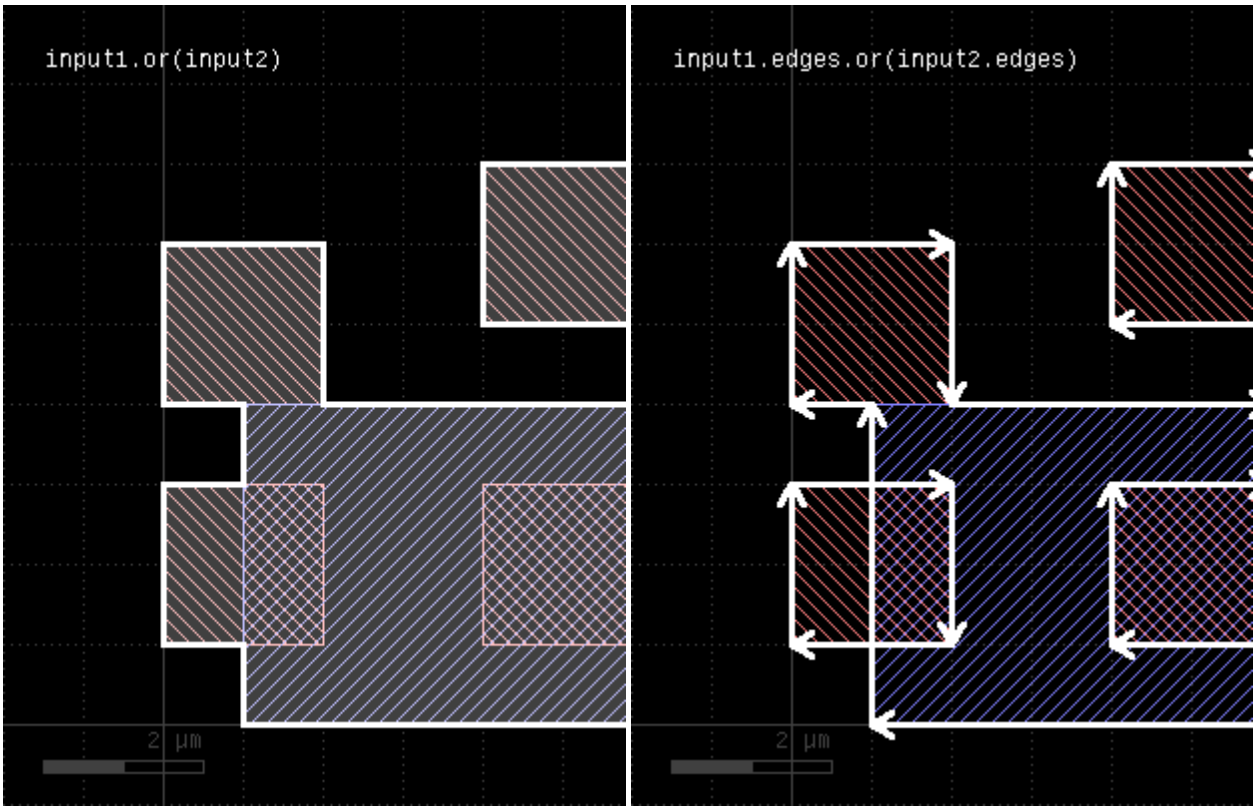
Usage:

- `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 (layer1: red, layer2: 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 global#report and global#target on how to configure output to a target layout or report database.

"outside" - Selects shapes or regions of self which are outside the other region

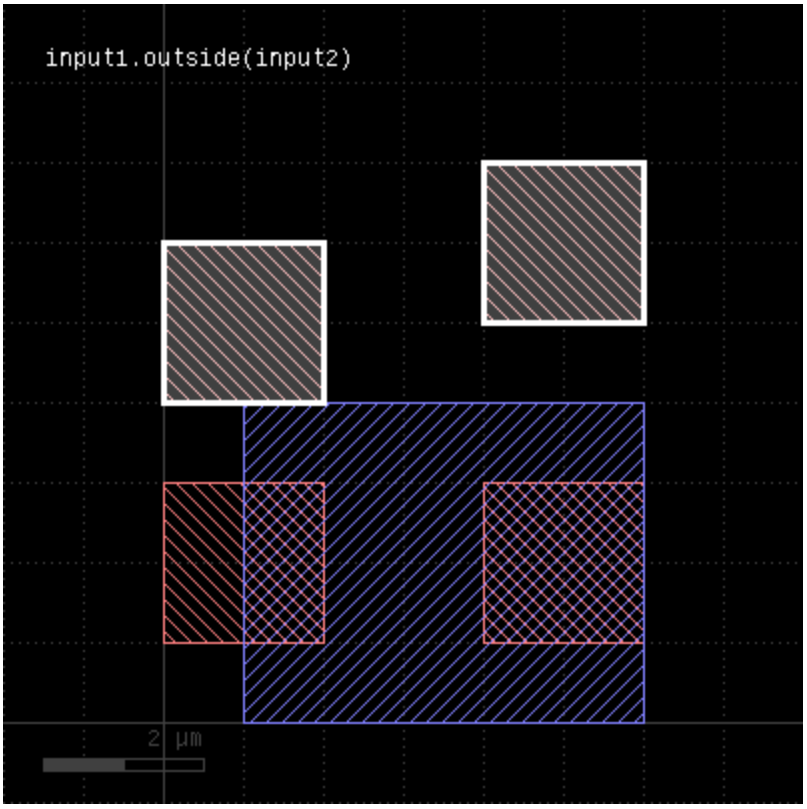
Usage:

- `layer.outside(other)`

This method selects all shapes or regions from self which are completely outside the other region (no part of these shapes or regions may be covered by shapes from the other region). If self is in raw mode, this method will select individual shapes. Otherwise, this method will select coherent regions and no part of these regions may overlap with shapes from the other region. It returns a new layer containing the selected shapes. A version which modifies self is select_outside.

This method is available for polygon layers.

The following image shows the effect of the "outside" method (input1: red, input2: blue):



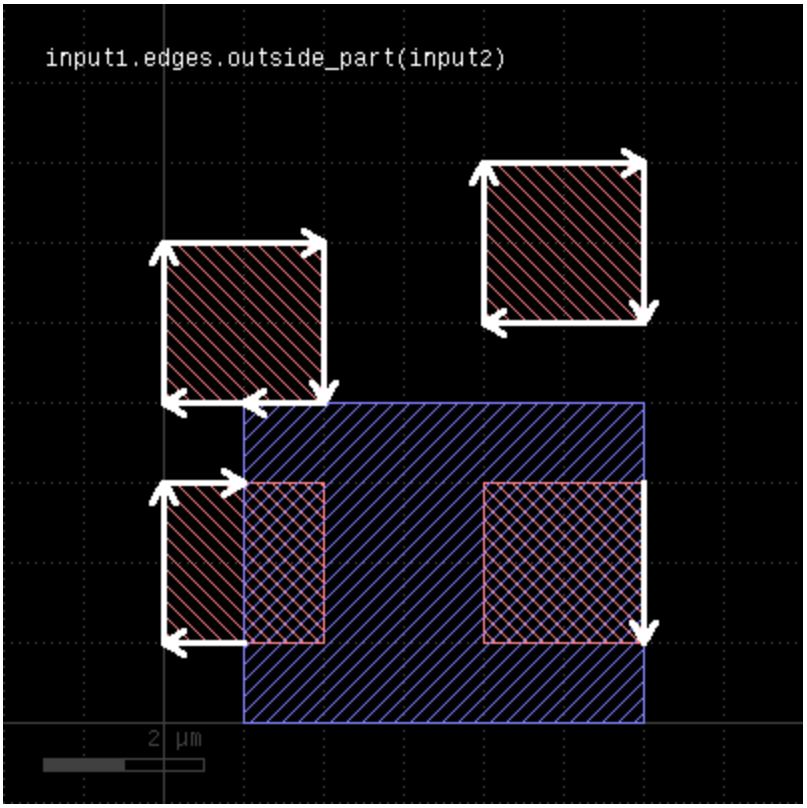
"outside_part" - Returns the parts of the edges outside the given region

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.



"overlap" - An overlap check

Usage:

- `layer.overlap(other_layer, value [, options])`

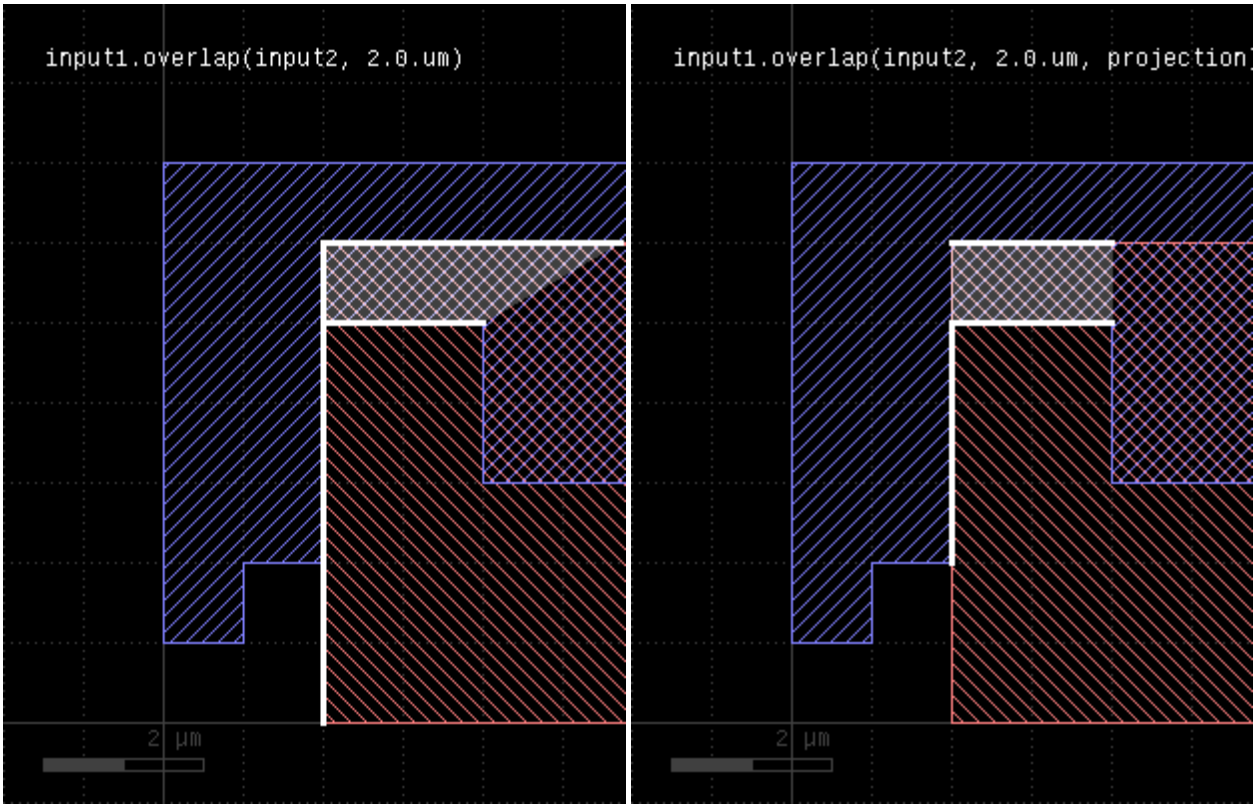
This method checks whether layer and other_layer overlap by at least the given length. Locations, where this is not the case will be reported in form of edge pair error markers. Locations, where both layers touch will be reported as errors as well. Formally such locations form an overlap with a value 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".

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. The options available are the same than for width. 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 (layer1: red, layer2: blue):



"overlapping" - Selects shapes or regions of self which overlap shapes from the other region

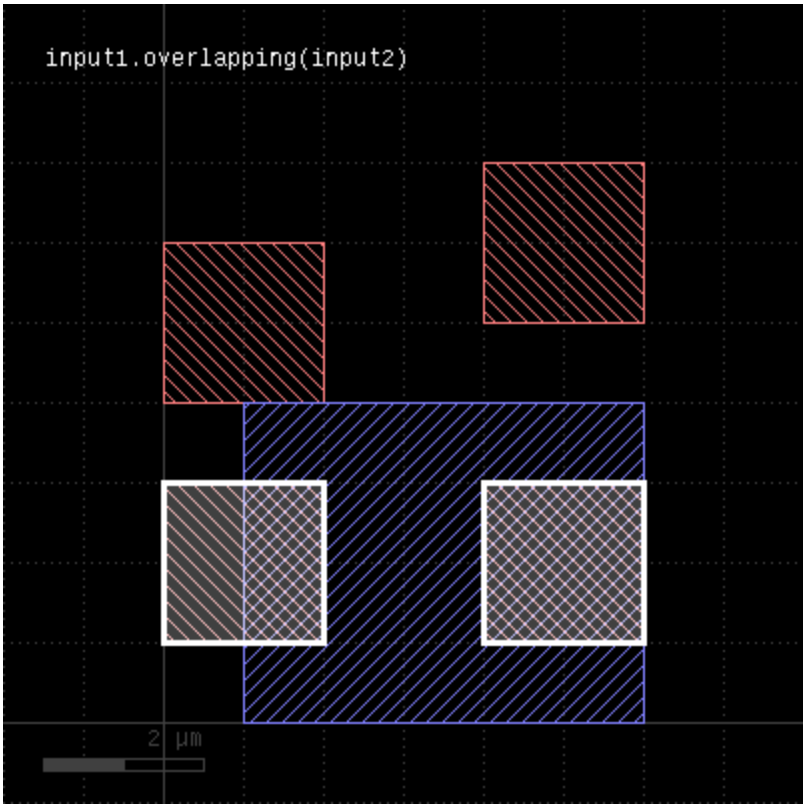
Usage:

- `layer.overlapping(other)`

This method selects all shapes or regions from self which overlap shapes from the other region. If 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 polygon and edge layers. Edges can be selected with respect to other edges or polygons.

The following image shows the effect of the "overlapping" method:



"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?`

"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 rectangle polygons 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.

"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.

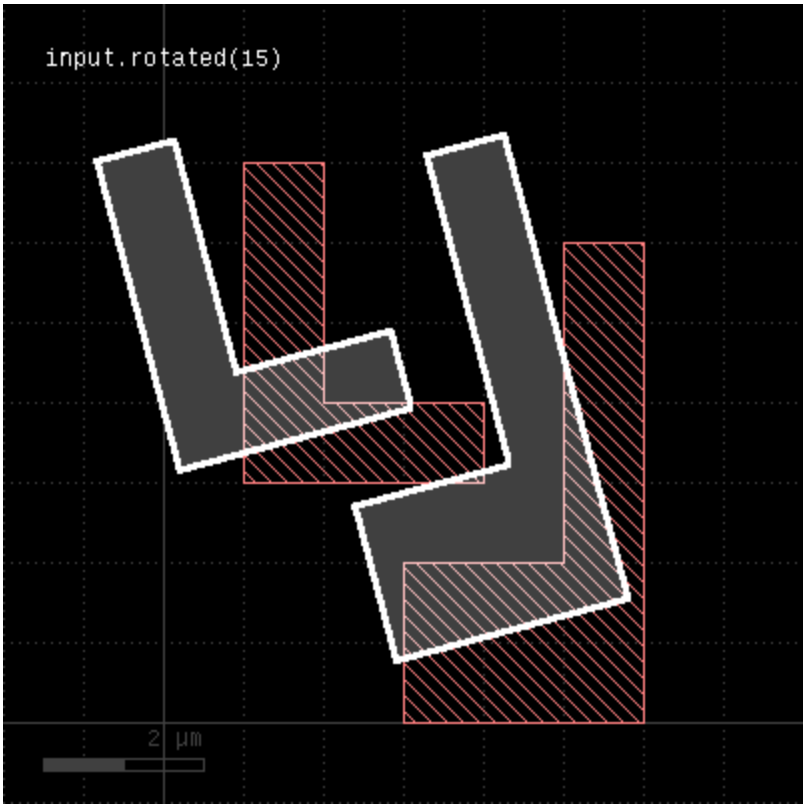
"rotated" - Rotates a layer

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:

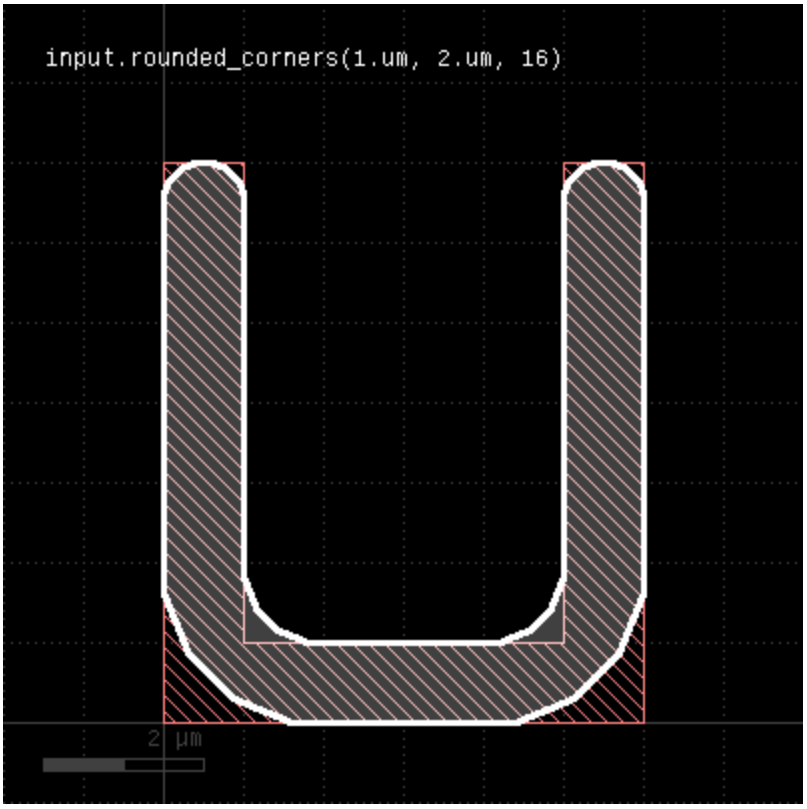
- `layer.rounded_corners(inner, outer, n)`

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.

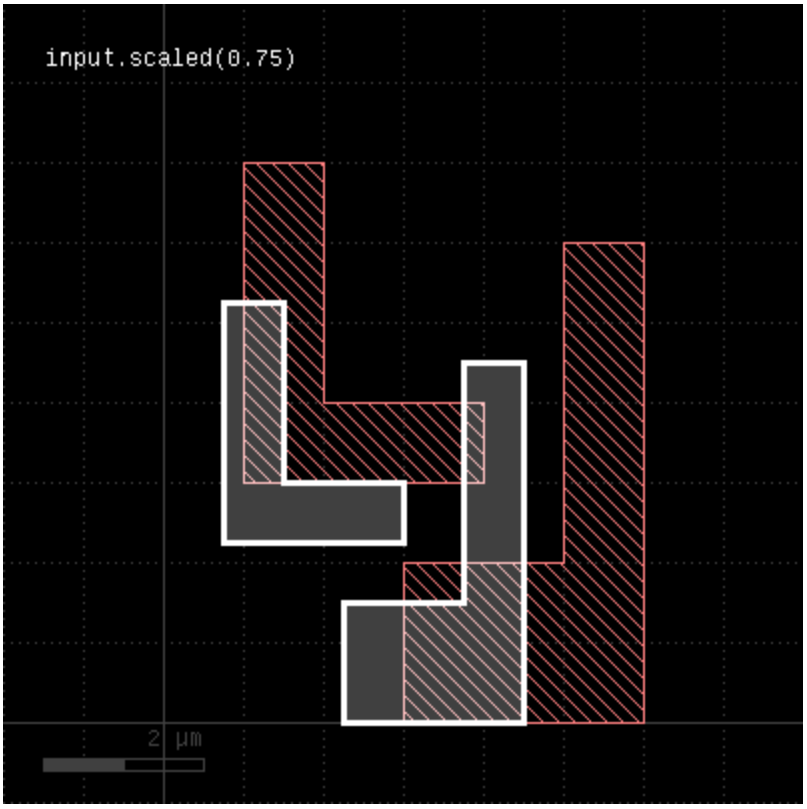
"scaled" - Scales a layer

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.

"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_inside" - Selects shapes or regions of self which are inside the other region

Usage:

- `layer.select_inside(other)`

This method selects all shapes or regions from self which are inside the other region. completely (completely covered by polygons from the other region). If self is in raw mode, this method will select individual shapes. Otherwise, this method will select coherent regions and no part of these regions may be outside the other region. It modifies self to contain the selected shapes. A version which does not modify self is inside.

This method is available for polygon layers.

"select_interacting" - Selects shapes or regions of self which touch or overlap shapes from the other region

Usage:

- `layer.select_interacting(other)`

This method selects all shapes or regions from self which touch or overlap shapes from the other region. If 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 and edge layers. Edges can be selected with respect to other edges or polygons.

"select_not_inside" - Selects shapes or regions of self which are not inside the other region

Usage:

- `layer.select_not_inside(other)`

This method selects all shapes or regions from self which are not inside the other region. completely (completely covered by polygons from the other region). If self is in raw mode, this method will select individual shapes. Otherwise, this method will select coherent regions and no part of these regions may be outside the other region. It modifies self to contain the selected shapes. A version which does not modify self is not_inside.

This method is available for polygon layers.

"select_not_interacting" - Selects shapes or regions of self which do not touch or overlap shapes from the other region

Usage:

- `layer.select_interacting(other)`

This method selects all shapes or regions from self which do not touch or overlap shapes from the other region. If 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 and edge layers. Edges can be selected with respect to other edges or polygons.

"select_not_outside" - Selects shapes or regions of self which are not outside the other region

Usage:

- `layer.select_not_outside(other)`

This method selects all shapes or regions from self which are not completely outside the other region (part of these shapes or regions may be covered by shapes from the other region). If self is in raw mode, this method will select individual shapes. Otherwise, this method will select coherent regions and no part of these regions may overlap with shapes from the other region. It modifies self to contain the selected shapes. A version which does not modify self is not_outside.

This method is available for polygon layers.

"select_not_overlapping" - Selects shapes or regions of self which do not overlap shapes from the other region

Usage:

- `layer.select_not_overlapping(other)`

This method selects all shapes or regions from self which do not overlap shapes from the other region. If 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 polygon and edge layers. Edges can be selected with respect to other edges or polygons.

"select_outside" - Selects shapes or regions of self which are outside the other region

Usage:

- `layer.select_outside(other)`

This method selects all shapes or regions from self which are completely outside the other region (no part of these shapes or regions may be covered by shapes from the other region). If self is in raw mode, this method will select individual shapes. Otherwise, this method will select coherent regions and no part of these regions may overlap with shapes from the other region. It modifies self to contain the selected shapes. A version which does not modify self is outside.

This method is available for polygon layers.

"select_overlapping" - Selects shapes or regions of self which overlap shapes from the other region

Usage:

- `layer.select_overlapping(other)`

This method selects all shapes or regions from self which overlap shapes from the other region. If 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 polygon and edge layers. Edges can be selected with respect to other edges or polygons.

"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:

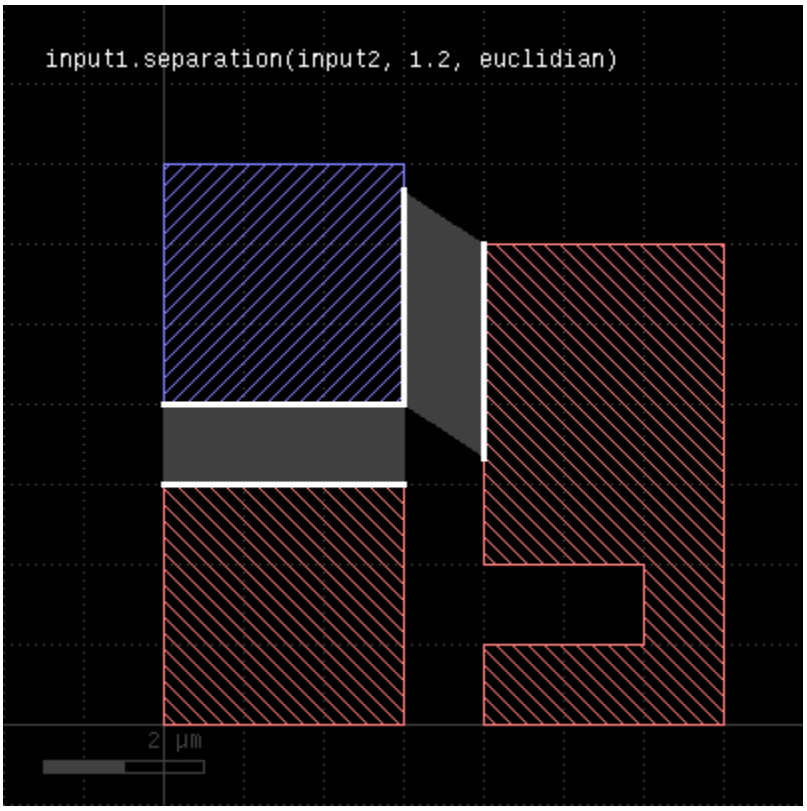
- `layer.separation(other_layer, value [, options])`

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.

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. The options available are the same than for width. 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 (layer1: red, layer2: blue):



"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

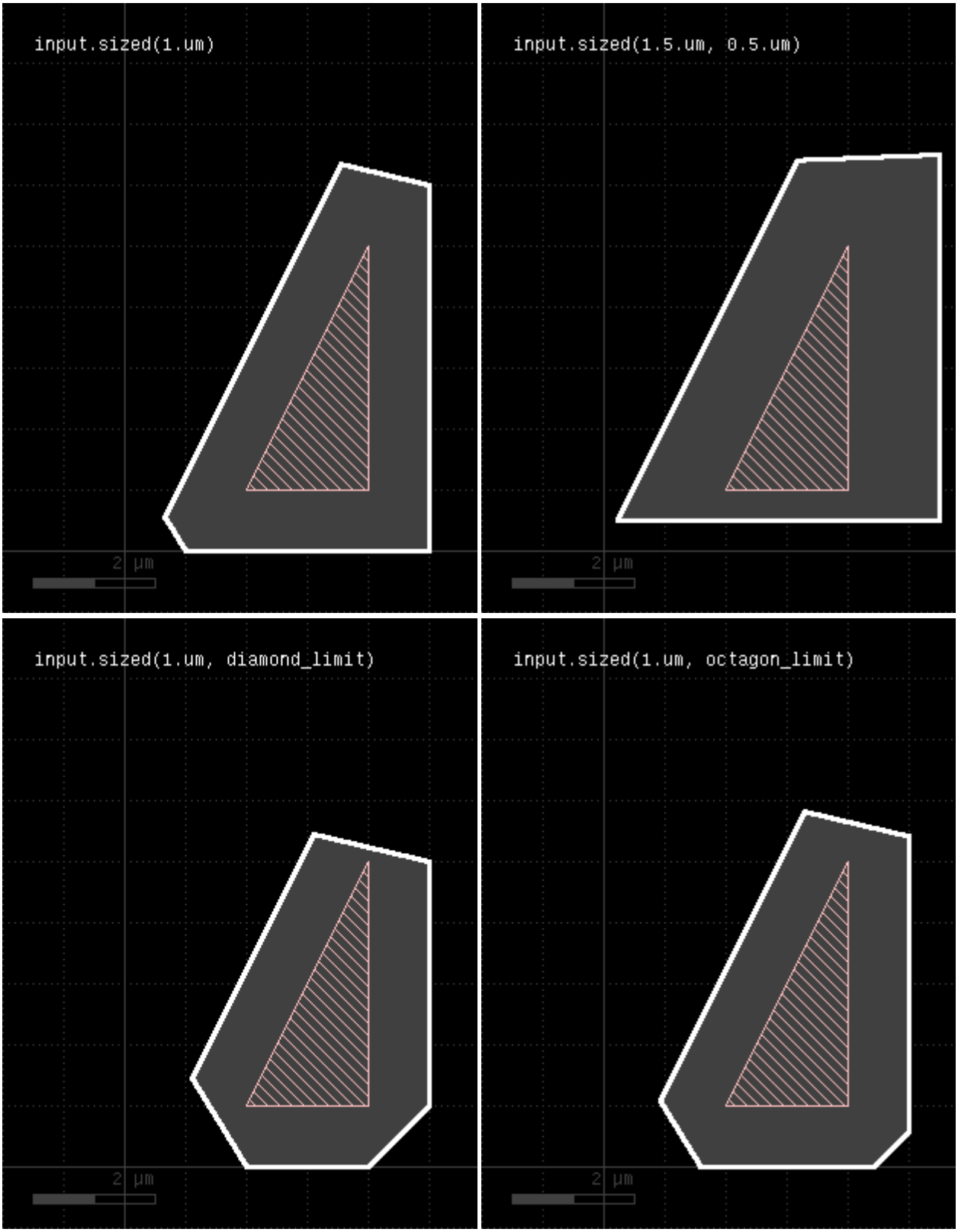
Merged semantics applies, i.e. polygons will be merged before the sizing is applied unless the layer was put into raw mode (see raw). On output, the polygons are not merged immediately, so it is possible to detect overlapping regions after a positive sizing using raw and merged with an overlap count, for example:

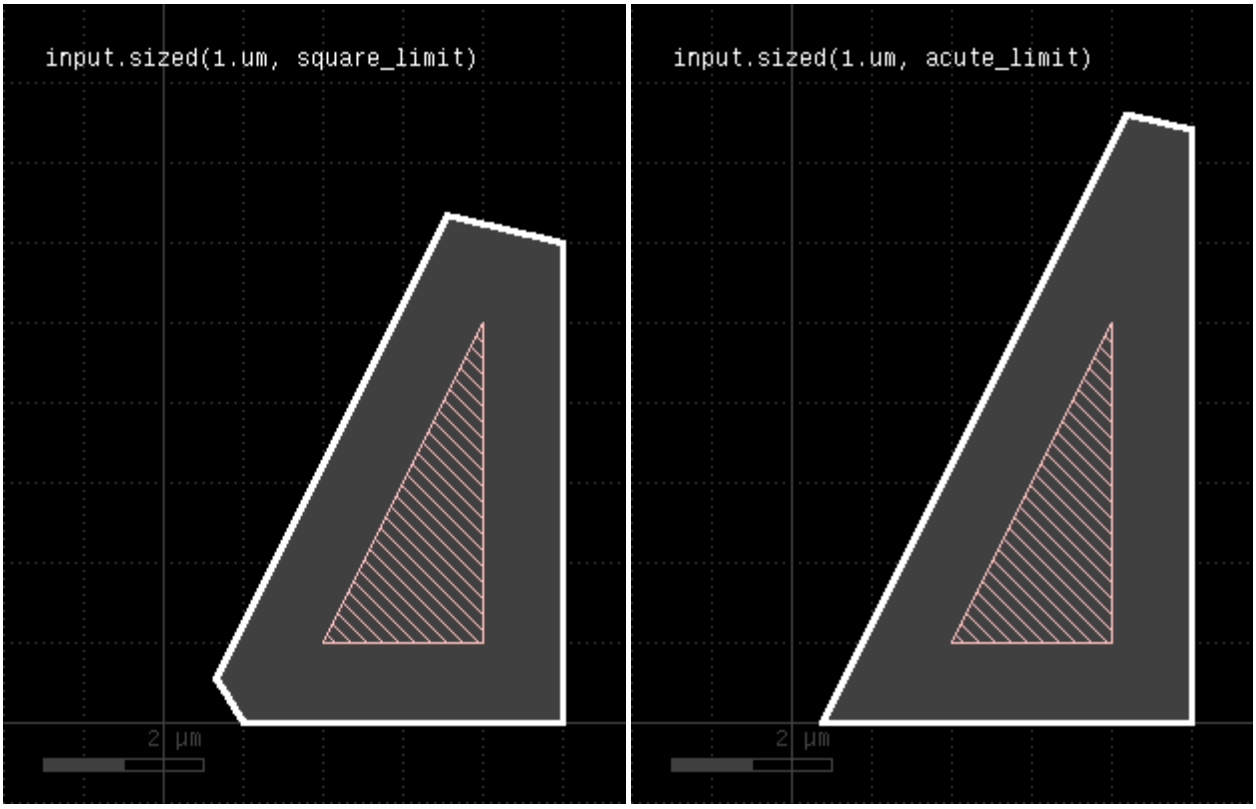
```
layer.sized(300.nm).raw.merged(2)
```

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)`

"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.

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:

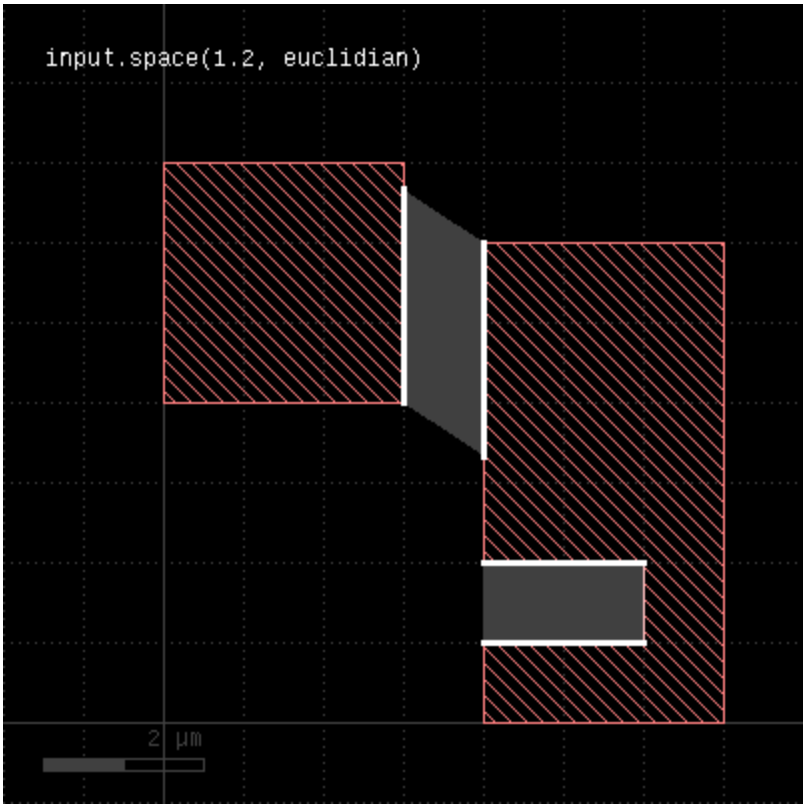
- `layer.space(value [, options])`

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.

The options available are the same than for the width method. Like for the width method, 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 space check:



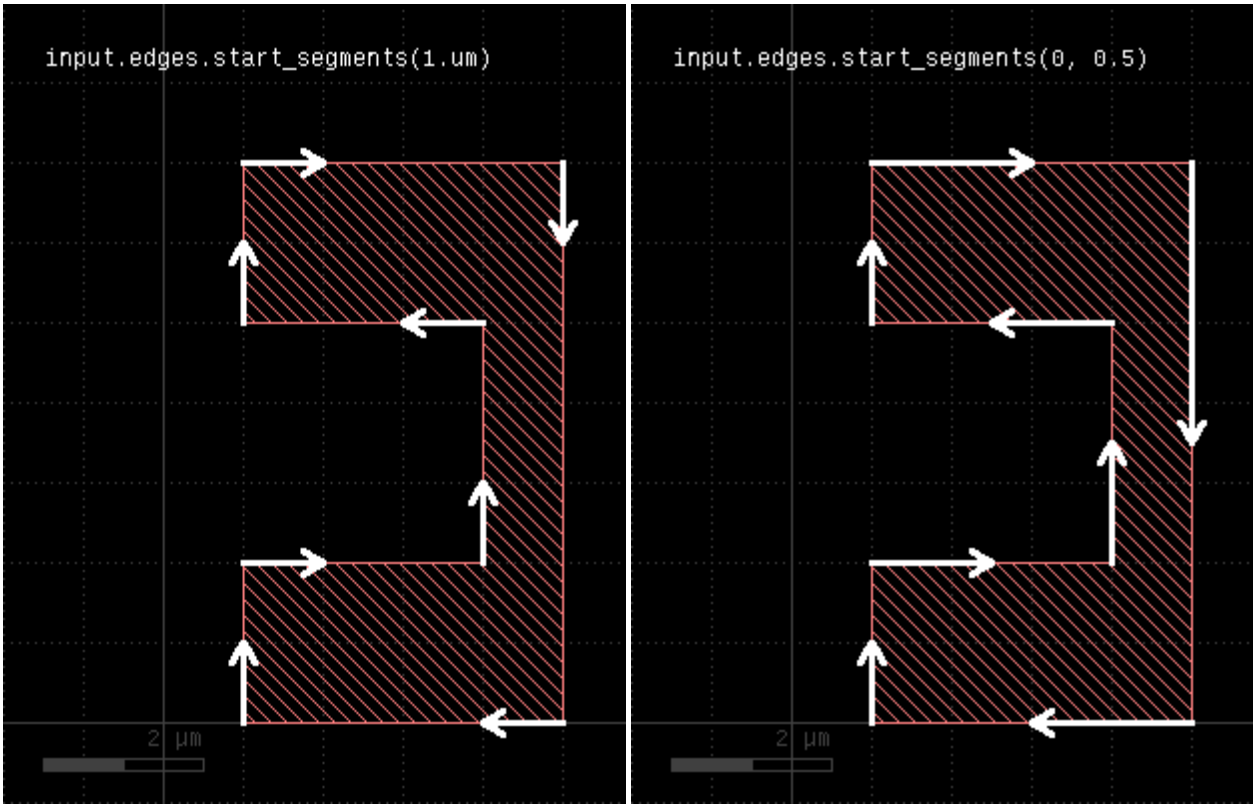
"start_segments" - Returns the part at the beginning of each edge

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.

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 = input(1, 0).texts
# Selects all texts beginning with an "A"
t = input(1, 0).texts("A*")
t = input(1, 0).texts(pattern("A*"))
# Selects all texts whose string is "A*"
t = input(1, 0).texts(text("A*"))
```

"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.

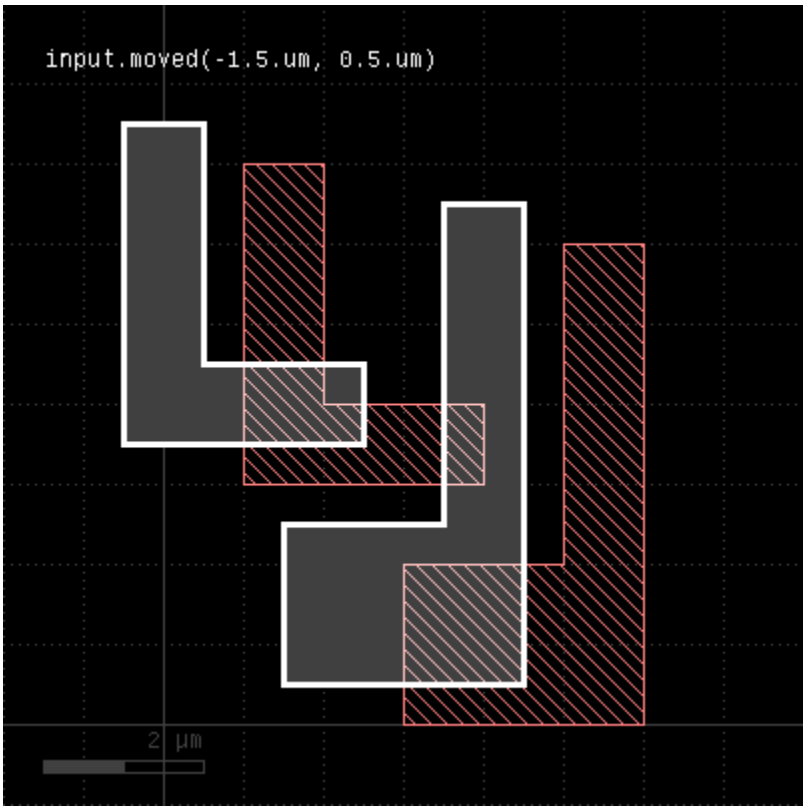
"transformed" - Transforms a layer

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])`

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.

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

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.

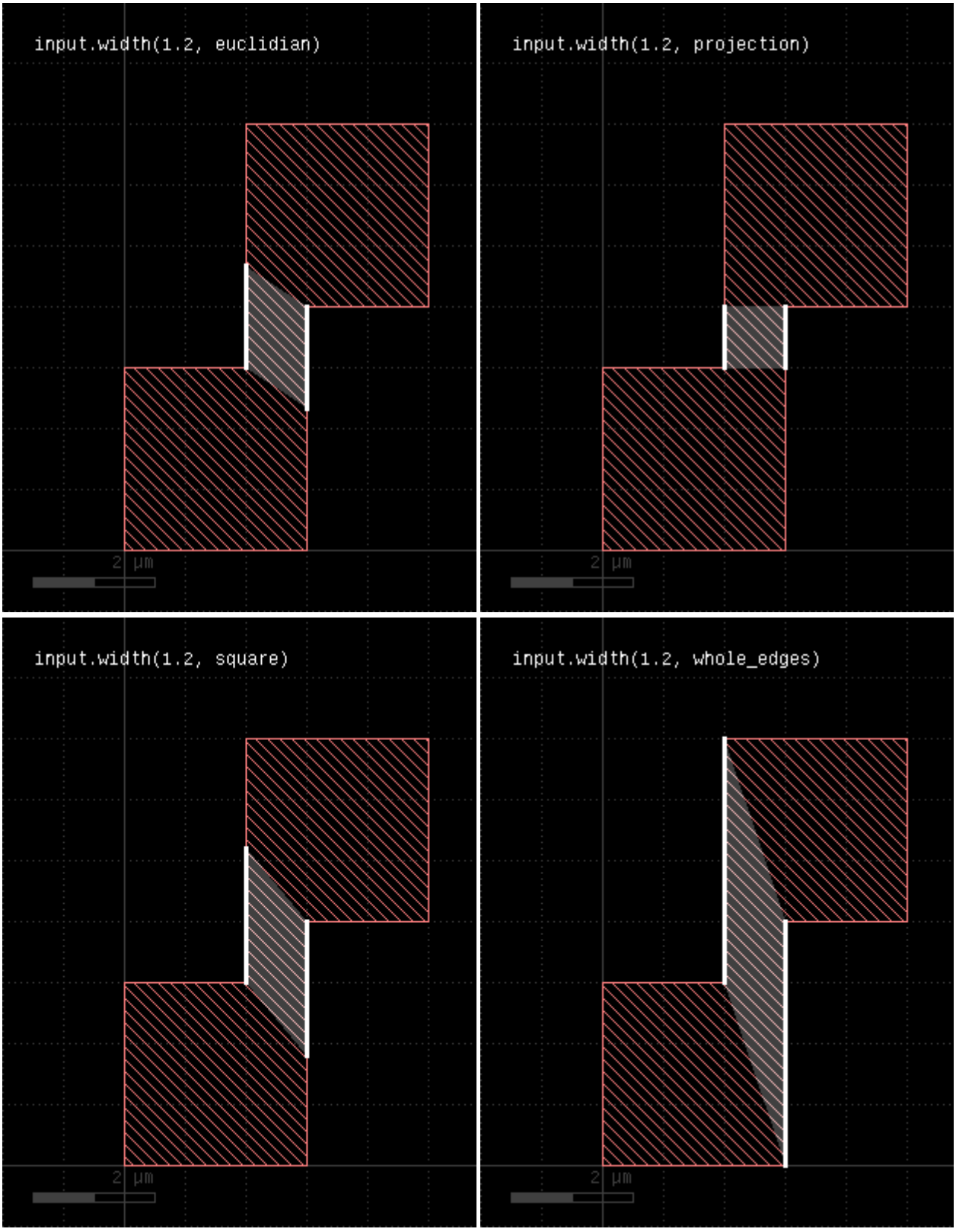
Merge semantics applies to this method, i.e. disconnected polygons are merged before the width is checked unless "raw" mode is chosen.

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)
```

The following images show the effect of various forms of the width check:



"with_angle" - Selects edges by their angle

Usage:

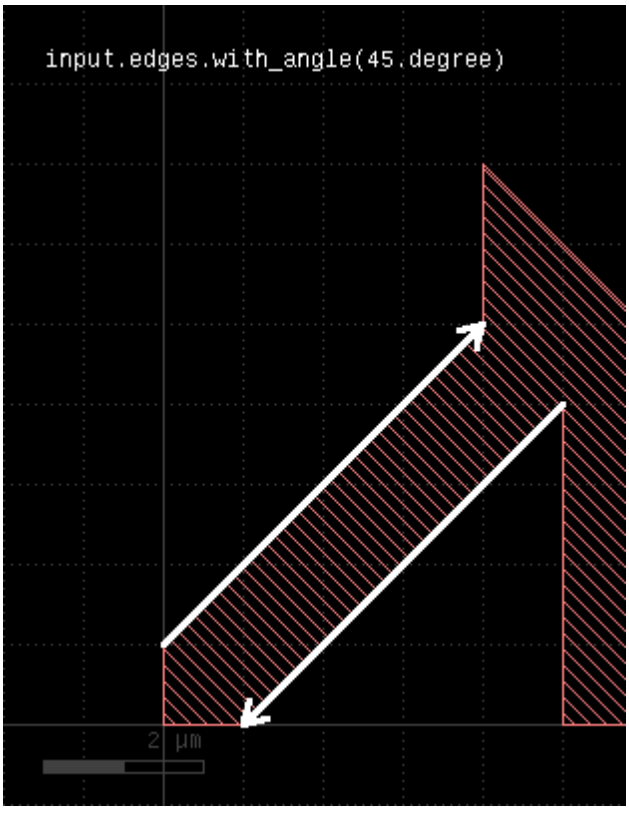
- `layer.with_angle(min .. max)`
- `layer.with_angle(value)`
- `layer.with_angle(min, max)`

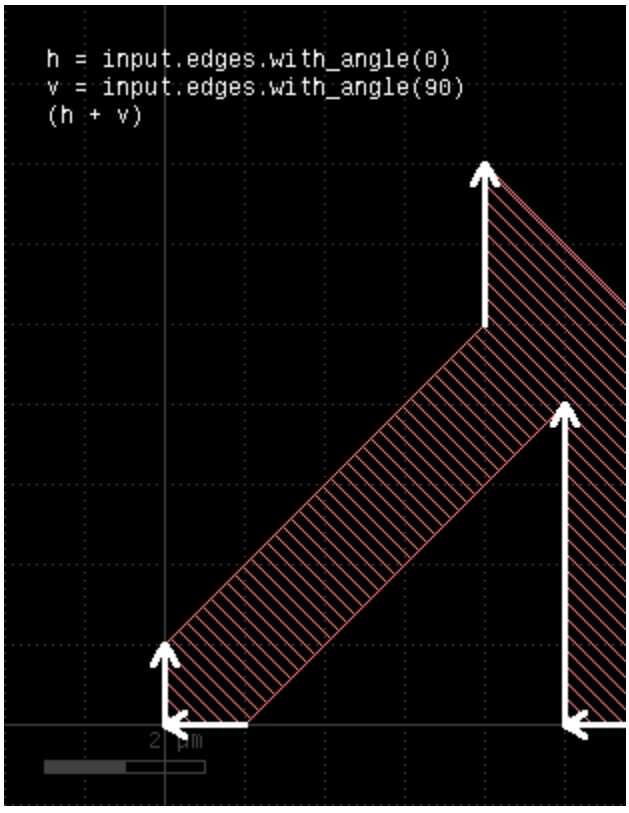
When called on an edge layer, the method selects edges by their angle, measured against the horizontal axis in the mathematical sense. The first version selects edges with a angle larger or equal to min and less than max (but not equal). The second version selects edges with exactly the given angle. The third version is identical to the first one.

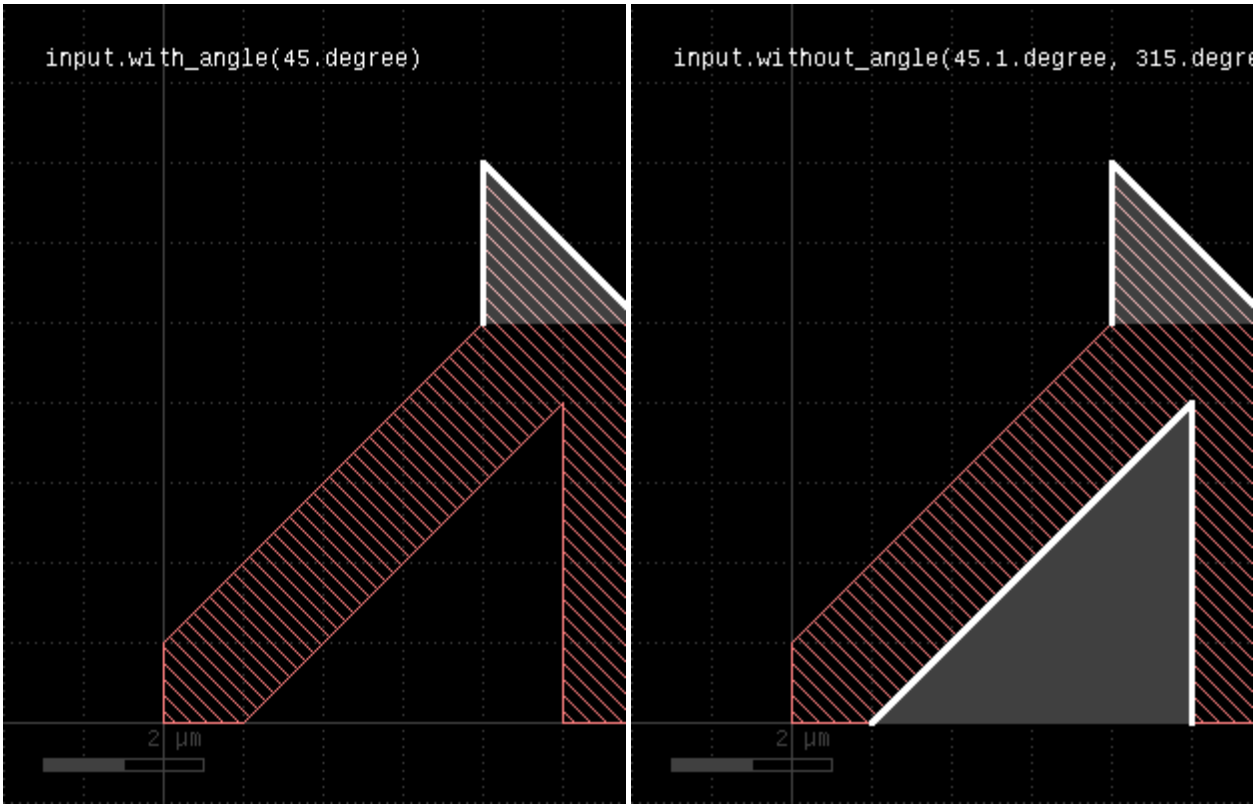
When called on a polygon layer, this method selects corners which match the given angle or is within the given angle interval. The angle is measured between the edges forming the corner. For each corner, an edge pair containing the edges forming in the angle is returned.

A method delivering all objects not matching the angle criterion is without_angle.

The following images demonstrate some use cases of with_angle and without_angle:







"with_area" - Selects polygons by area

Usage:

- `layer.with_area(min .. max)`
- `layer.with_area(value)`
- `layer.with_area(min, max)`

The first form will select all polygons with an area larger or equal to min and less (but not equal to) max. The second form will select the polygons 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.

"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_length" - Selects edges by their length

Usage:

- `layer.with_length(min .. max)`
- `layer.with_length(value)`
- `layer.with_length(min, max)`

The method selects edges by their length. The first version selected 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 layers only.

"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.

"without_angle" - Selects edges by the their angle

Usage:

- `layer.without_angle(min .. max)`
- `layer.without_angle(value)`
- `layer.without_angle(min, max)`

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).

"without_area" - Selects polygons 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 without an area equal to the given one or outside the given interval.

This method is available for polygon layers only.

"without_bbox_height" - Selects polygons by the height of the bounding box

Usage:

- `layer.without_bbox_height(min .. max)`
- `layer.without_bbox_height(value)`
- `layer.without_bbox_height(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_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_length" - Selects edges by the their length

Usage:

- `layer.without_length(min .. max)`
- `layer.without_length(value)`
- `layer.without_length(min, max)`

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 layers only.

"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.

"xor" - Boolean XOR operation

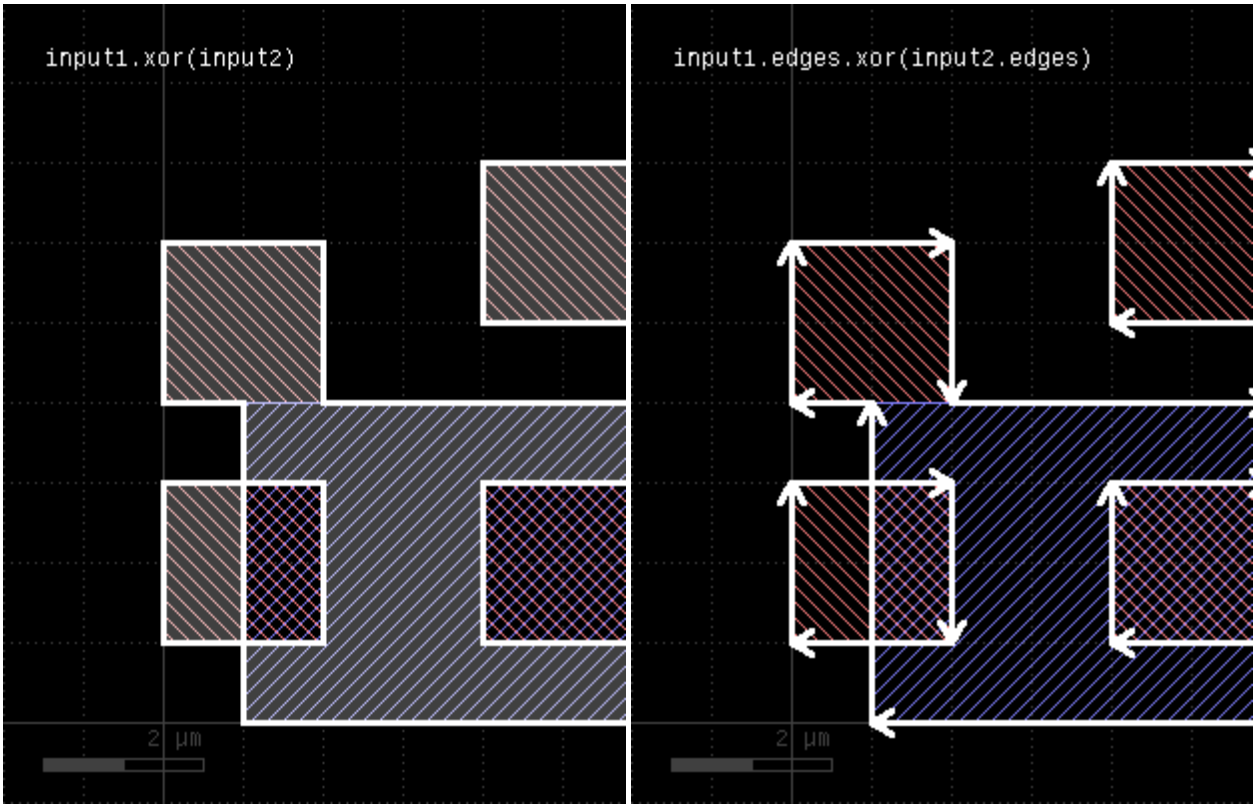
Usage:

- `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 (layer1: red, layer2: 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.

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 ... ])`

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)
```

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.

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.

"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 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.

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_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.

"connect_implicit" - Specifies a search pattern for labels which create implicit net connections

Usage:

- `connect_implicit(label_pattern)`

Use this method to supply label strings which create implicit net connections on the top level circuit. This feature is useful to connect identically labelled nets while a component isn't integrated yet. If the component is integrated, nets may be connected on a higher hierarchy level - e.g. by a power mesh. Inside the component this net consists of individual islands. To properly perform netlist extraction and comparison, these islands need to be connected even though there isn't a physical connection. "connect_implicit" can achive this if these islands are labelled with the same text on the top level of the component.

The implicit connections are applied on the next net extraction and cleared on "clear_connections".

"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.

Each device class (e.g. n-MOS/p-MOS or high Vt/low Vt) needs it's 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 })
```

"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.

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.

"extent" - Returns a layer with the bounding box of the selected layout

Usage:

- `source.extent`

The extent function is useful to invert a layer:

```
inverse_1 = extent.sized(100.0) - input(1, 0)
```

"input" - Specifies input from a source

Usage:

- `source.input(layer)`
- `source.input(layer, datatype)`
- `source.input(layer_into)`
- `source.input(filter, ...)`

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" contain both texts 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.

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(layer)`
- `source.labels(layer, datatype)`
- `source.labels(layer_into)`
- `source.labels(filter, ...)`

Creates a 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.

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.

"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(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.

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:

- `source.select(filter1, filter2, ...)`

This method will construct a new source object with the given cell filters applied. 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:

```
layout_with_selection = layout.select("-TOP", "+B")
l1 = layout_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". The following code will just select "B" without it's children, because in the first "-*" selection, all cells including the children of "B" are disabled:

```
layout_with_selection = layout.select("-*", "+B")
l1 = layout_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)

DRC 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.

"antenna_check" - Performs an antenna check

Usage:

- `antenna_check(gate, metal, ratio, [ diode_specs ... ])`

See Netter#antenna_check for a description of that function.

"bjt3" - Supplies the BJT3 transistor extractor class

Usage:

- `bjt3(name)`

Use this class with extract_devices to specify extraction of a bipolar junction transistor

"bjt4" - Supplies the BJT4 transistor extractor class

Usage:

- `bjt4(name)`

Use this class with extract_devices to specify extraction of a bipolar junction transistor with a substrate terminal

"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)`

Use this class with extract_devices to specify extraction of a capacitor. The area_cap argument is the capacitance in Farad per square micrometer.

"capacitor_with_bulk" - Supplies the capacitor extractor class that includes a bulk terminal

Usage:

- `capacitor_with_bulk(name, area_cap)`

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.

"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)
```

"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)
```

"connect" - Specifies a connection between two layers

Usage:

- `connect(a, b)`

See Netter#connect for a description of that function.

"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)`

See Netter#connect_implicit for a description of that function.

"dbu" - Gets or sets the database unit to use

Usage:

- `dbu(dbu)`
- `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.

"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)`

Use this class with extract_devices to specify extraction of a planar diode

"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.

"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

Usage:

- `extent`

See Source#extent for a description of that function.

"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.

"info" - Outputs as message to the logger window

Usage:

- `info(message)`

Prints the message to the log window in verbose mode. In non-verbose more, nothing is printed. 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.

"is_deep?" - Returns true, if in deep mode

Usage:

- `is_deep?`

"is_tiled?" - Returns true, if in tiled mode

Usage:

- `is_tiled?`

"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`

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.

"log" - Outputs as message to the logger window

Usage:

- `log(message)`

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.

"mos3" - Supplies the MOS3 transistor extractor class

Usage:

- `mos3(name)`

Use this class with extract_devices to specify extraction of a three-terminal MOS transistor

"mos4" - Supplies the MOS4 transistor extractor class

Usage:

- `mos4(name)`

Use this class with extract_devices to specify extraction of a four-terminal MOS transistor

"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

"no_borders" - Reset the tile borders

Usage:

- `no_borders`

Resets the tile borders - see tile_borders for a description of tile borders.

"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.

"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.

"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.

"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)`

Use this class with extract_devices to specify extraction of a resistor. The sheet_rho value is the sheet resistance in ohms/square.

"resistor_with_bulk" - Supplies the resistor extractor class that includes a bulk terminal

Usage:

- `resistor_with_bulk(name, sheet_rho)`

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.

"select" - Specifies cell filters on the default source

Usage:

- `select(args)`

See Source#select for a description of that function.

"silent" - Resets verbose mode

Usage:

- `silent`

This function is equivalent to "verbose(false)" (see verbose)

"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 *a 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.

"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" specifying output to a 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)`

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.

"tile_borders" - Specifies a minimum tile border

Usage:

- `tile_border(b)`
- `tile_border(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 (and, or, ...), size and the DRC functions (width, space, ...) a tile border is automatically established. For other operations such as with_area or 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).

"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

"write_spice" - Defines SPICE output format (with options)

Usage:

- `write_spice([ 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.