KLayout

Introduction into the Python API

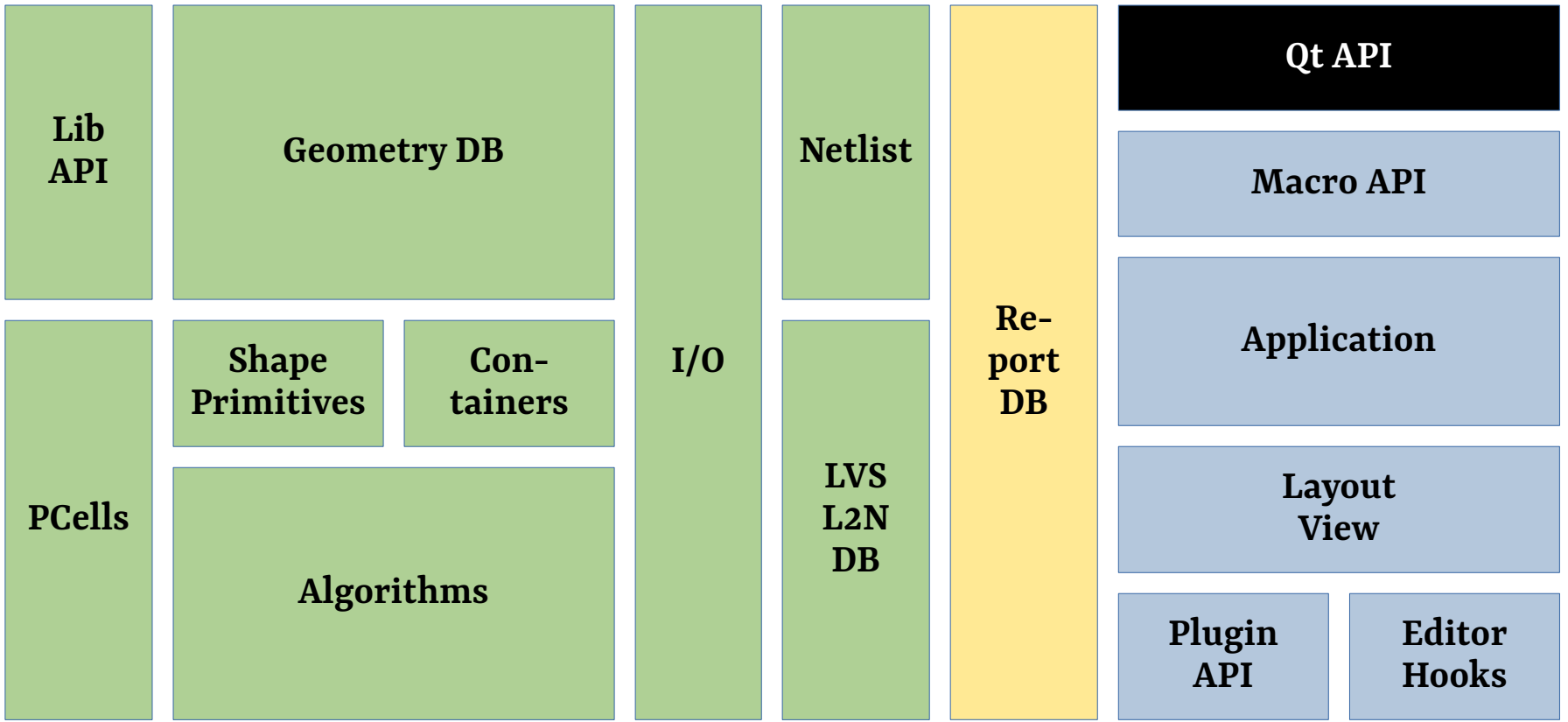
Matthias Köfferlein, www.klayout.org

Things you can do with the API

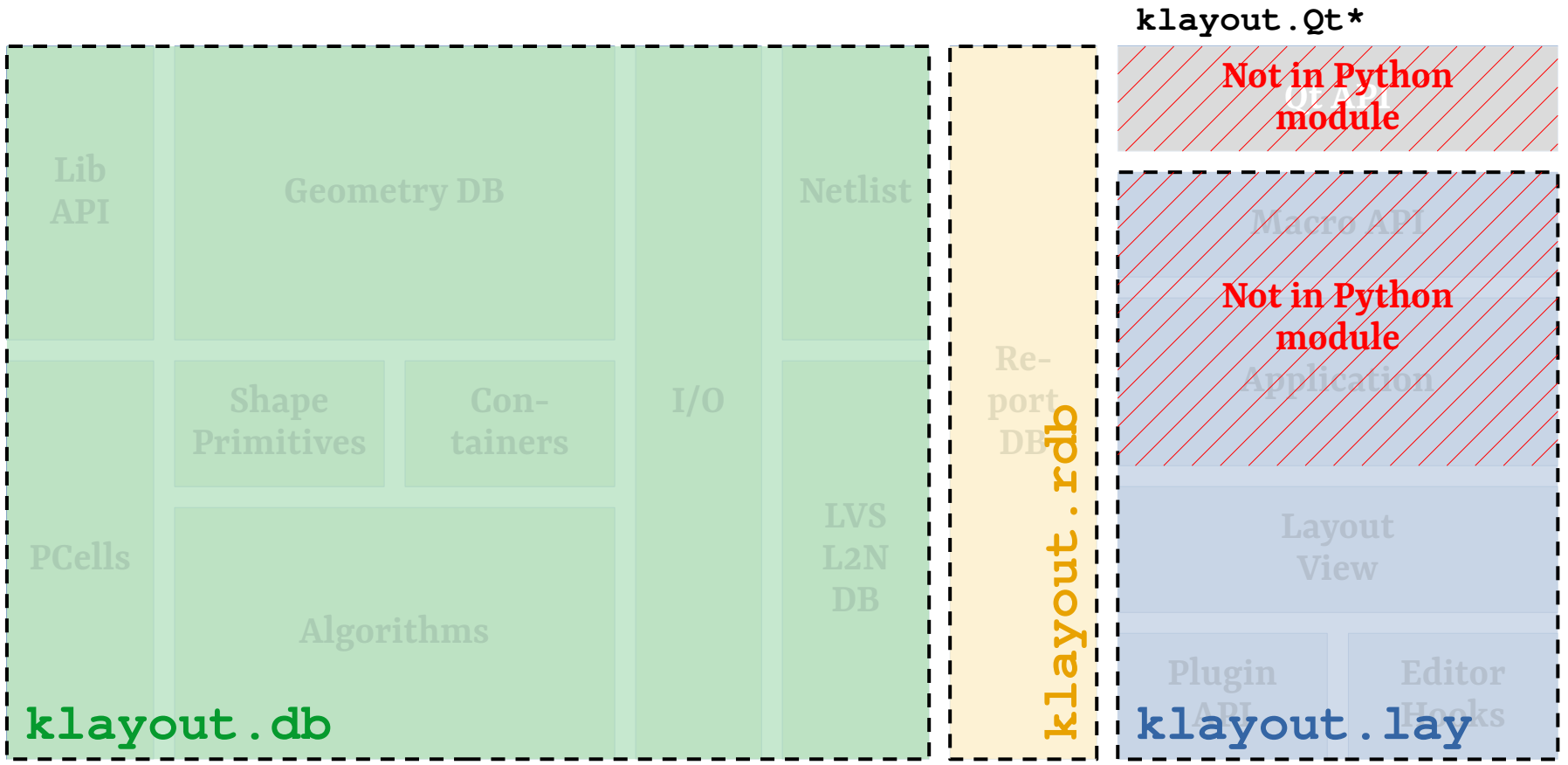
- Generate Layout
- Write PCells
- Analyze Layout (measure, verify)
- Manipulate Layout
- Extract Netlists
- Work with Netlists (compare, manipulate)
- Work with Report Databases
- Automate Tasks in the application
- Manipulate Layout views
- Write Plugins

And more …

An Brief Overview



Python Namespaces



How to use KLayout/Python

Option 1: “klayout” module on PyPI

- You can load, analyze and modify layouts, report database, netlists etc.
- Lightweight: No UI support, no Qt support
- LayoutView is available for rendering (no UI)

How to use KLayout/Python

Option 2: Inside the application

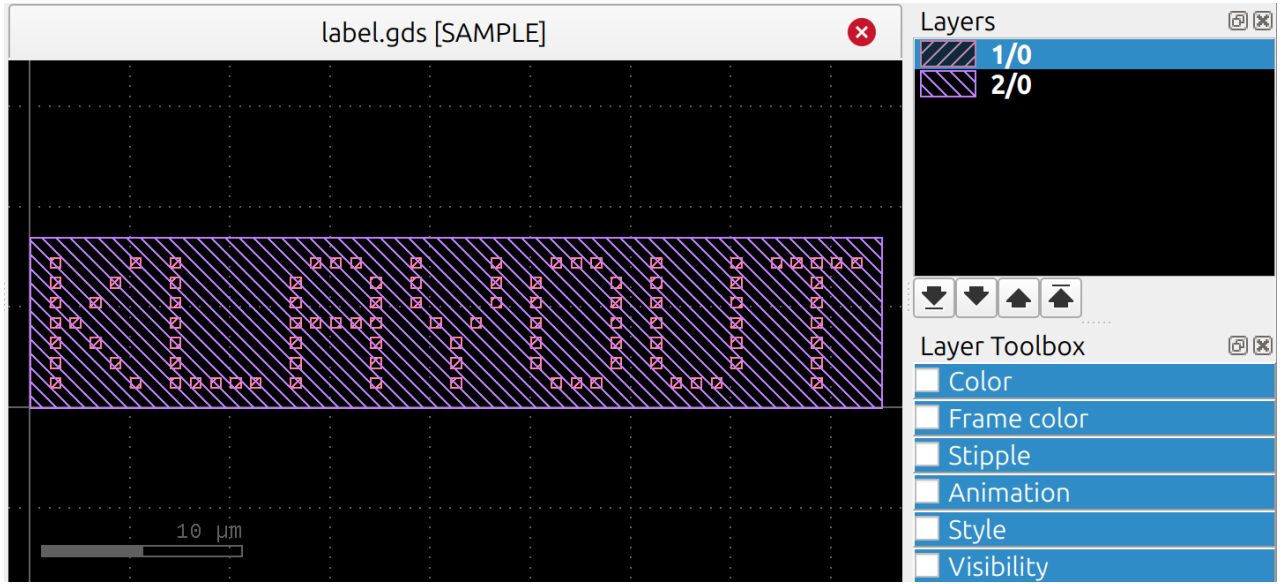
- All options are available
- You can run scripts
  - explicitly from the IDE
  - let them plug features into the application

Example script

```
import klayout.db as db

ly = db.Layout()

# sets the database unit to 1 nm
ly.dbu = 0.001

# adds a single top cell
top_cell = ly.create_cell("SAMPLE")

# creates a new layer
# (layer number 1, datatype 0)
layer1 = ly.layer(1, 0)

pattern = """
.#...#.#.....###..#...#..###..#...#.#####
.#..#..#.....#...#.#...#.#...#.#...#...#..
.#.#...#.....#...#.#...#.#...#.#...#...#..
.##....#.....#####..#.# .#...#.#...#...#..
.#.#...#.....#...#...#. .#...#.#...#...#..
.#..#..#.....#...#...#. .#...#.#...#...#..
.#...#.#####.#...#...#. ..###...###....#..
"""

# produces pixels from the bitmap as
# 0.5x0.5 µm boxes on a 1x1 µm grid:
y = 8.0
for line in pattern.split("\n"):
  x = 0.0
  for bit in line:
    if bit == "#":
      # creates a rectangle for the "on" pixel
      rect = db.DBox(0, 0, 0.5, 0.5).moved(x, y)
      top_cell.shapes(layer1).insert(rect)
    x += 1.0
  y -= 1.0

# adds an envelope box on layer 2/0
layer2 = ly.layer(2, 0)
envelope = top_cell.dbbox().enlarged(1.0, 1.0)
top_cell.shapes(layer2).insert(envelope)

# writes the layout to GDS
ly.write("label.gds")
```

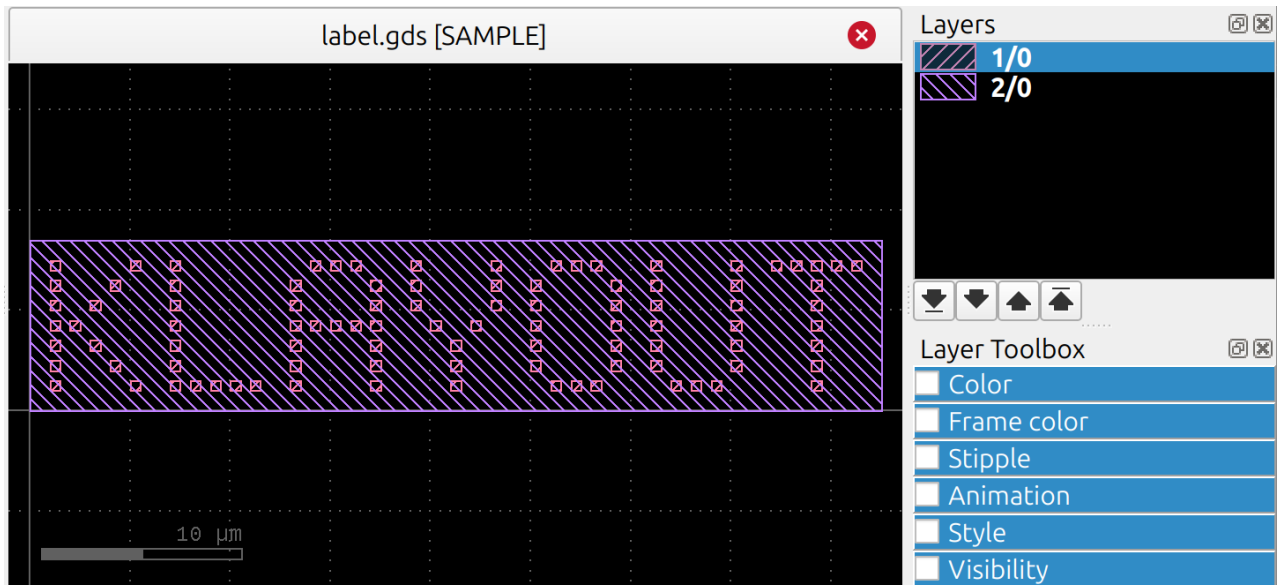
Running with Python module

```
$ pip3 install klayout
$ python3 python_script.py
$ klayout label.gds
```



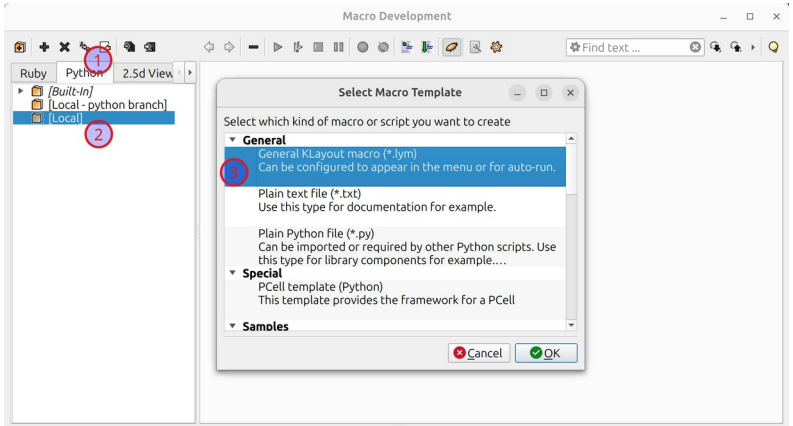
Running with KLayout (batch)

```
$ klayout -b -r python_script.py
$ klayout label.gds
```

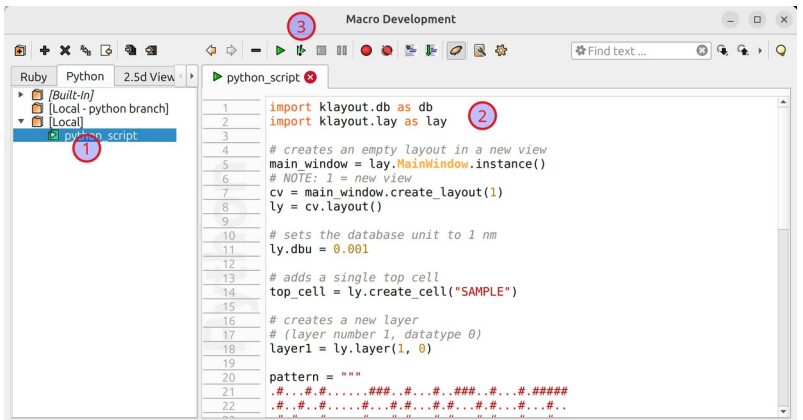


Running in KLayout

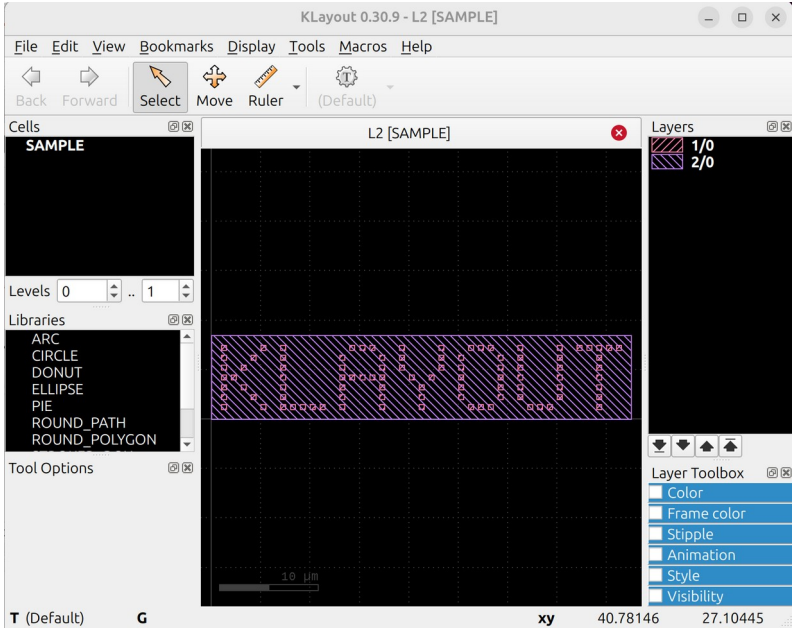
Menu: Macros → Macro Editor



Rename script, paste code and hit “run”



This is a different script version that opens the layout directly in a new tab



Tips

- “pya” is a deprecated alias for all of the “klayout.*” modules
- When running interactively from the IDE, there are two “run” buttons:
  -  Runs the “active” script
  -  Makes the current tab active and runs it
- You can use the debugger to set breakpoints and single-step through the scripts

Script spaces in the Application

- **Python scripts are stored in**
  - **$KLAYOUT_HOME/pymacros** for “lym” style scripts These are XML files with the script code and additional metadata – e.g. menu binding
  - **$KLAYOUT_HOME/python** for plain Python scripts This folder appears in **sys.path.** You can use this location to store your own Python modules for “import”
- **You can add new spaces with “Add Location” from the IDE’s context menu from the “Python” tab**

The Console

Use the Console in the Macro IDE to enter commands and execute them immediately

- Make sure, “Python” mode is selected
- Runs on interpreter top level: beware of side effects

There can only be one …

- The application runs a single Interpreter
  - Currently there is no reset, except restarting
  - Imported modules become cached – when you test your own modules, that may become tedious
  - Classes defined in scripts remain in the interpreter space and will be redefined on re-executing the script
  - There can be side effects

Managed and unmanaged objects

- Object lifetime needs to be negotiated between Python and C++
  - Objects living in C++ must not get deleted by Python (“unmanaged”)
  - Objects managed by Python can be deleted and C++ has to know about that (such objects are called “managed”)
  - Ownership can move between Python and C++
- Some lightweight objects are simply copied (e.g. Box, Polygon etc.)
  - Python does not have a reference to them and modifying them does not have an effect
- Other objects are equipped with management data (the “heavy” ones)
  - Objects deleted in C++ will invalidate the Python object
  - Objects managed in Python space are removed in C++ space if they are deleted
  - Python keeps references to them you can modify

Methods vs. properties

- Python needs to have brackets after a method to actually call it
- Without that, you get a method object and strange errors
- And there are properties (get, set) which don't need this
- This can be confusing:
  - shape.bbox(): a method call
  - shape.box: a property getter
- This is how the documentation tells you it is a property:

Python specific notes:
The object exposes a readable attribute 'box'. This is the getter.

Resources

- PyPI module documentation
  https://www.klayout.org/klayout-pypi/
- Built-in Help System of KLayout
- Introduction into scripting in KLayout
  https://www.klayout.de/doc-qt5/programming/index.html
- Reference documentation
  https://www.klayout.de/doc-qt5/code/index.html
- Python specifics
  https://www.klayout.de/doc-qt5/programming/python.html

The Geometry Database

The Layout object

- The Layout object represents one stream file
- It is a collection of cells, layers with shapes and meta data
  - most importantly the database unit (DBU)
- Cells form a hierarchy graph through cell instances
  - A tree, if you look at every cell instance individually
- A cell without a parent is a “top cell”
  - Usually there is one top cell, but there can be many

The Layout object

```
# Creates a new empty Layout
# object

import klayout.db as db

ly = db.Layout()
ly.dbu = 0.001
```

```
# Gets the Layout from the currently
# loaded, active layout

import klayout.db as db
import klayout.lay as lay

ly = lay.CellView.active().layout()
```

“Under construction” mode

- Once a Layout is modified, it needs to update internal information
- In update loops this can create a significant overhead
- It is possible therefore to temporarily disable these updates:

```
# "ly" is a db.Layout object

try:
  ly.start_changes()
  # ly.under_construction() -> True
  ... do something ...
finally:
  ly.end_changes()
```

Editable and non-editable Layouts

- An “editable” Layout supports manipulations (i.e. delete/replace) better
  - Takes more memory
  - All features are available

```
import klayout.db as db

# Creates an editable layout
ly = db.Layout()
# or
ly = db.Layout(True)
```

- A “non-editable” Layout is mainly for read-only access
  - Used in read/analyze scenarios
  - More memory efficient
  - No warranty upon manipulations

```
import klayout.db as db

# Creates a non-editable layout
ly = db.Layout(False)
```

Working copies and DB objects

- The Geometry DB stores certain objects internally in a way not suitable for Python
- Example: A rectangle (Box) as stored as a very compact object with 16 Bytes and no context – cannot be bound to a Python object
- So when using a Box object in Python code, you use a working copy of the DB object
- Manipulation of DB objects goes like this:
  - Obtain a working copy
  - Manipulate it
  - Replace the DB object by the working copy
- In the same way, the DB is populated feeding working copy objects

DB object references

- To refer to a DB object, references are provided
- These act like pointers into the DB
  - The Shape object is a reference to a DB shape (box, polygon, path, text, ...)
  - The Instance object is a reference to a DB cell instance
- Shape and Instance references can become invalid during operations
  - Specifically in non-editable mode
  - When they point to a object that was removed

KLayout is not very pythonic

- Do not expect the usual Python behavior
- For example: inserting a working copy object into a Shapes container creates a copy

```
shapes = db.Shapes()

# insert a box
box = db.Box(0, 0, 10, 20)
shapes.insert(box)

print(box)
# -> (0,0;10,20)
print(";".join([ str(s) for s in shapes.each()]))
# -> box (0,0;10,20)

# changing the box does not change the content of
# the shape container
box.left = -10
print(box)
# -> (-10,0;10,20)
print(";".join([ str(s) for s in shapes.each()]))
# -> box (0,0;10,20)
```

Integer- and Float-type objects

- There are two types of working copy objects
  - Integer coordinates: Box, Polygon, Text, Path, CellInstArray, ...
  - Floating-point coordinates: DBox, DPolygon, DText, DPath, DCellInstArray, ...
- By convention, the floating-point coordinate types represent objects with micrometer-unit coordinates
- The integer-type objects represent objects with coordinates in DBU
  - Internally, the DB uses integer-type objects

Working copy objects

DB object	Integer-Type working copy	Float-Type working copy	Comment
Instance	**CellInstArray**	**DCellInstArray**	
Shape (depending on type)	**Box**	**DBox**	
	Polygon	**DPolygon**	
	SimplePolygon	**DSimplePolygon**	GDS2/OASIS compatible
	Path	**DPath**	
	Text	**DText**	
	Edge	**DEdge**	Not for GDS2 or OASIS
	EdgePair	**DEdgePair**	

Layouts and Layers

- Layers are not properties of shapes, but “slots”, the shapes are sorted into
- Layers are identified by an ID (an integer)
- The IDs are managed by the Layout object

```
# "ly" is a db.Layout object

# Gets the ID for layer 1/0
# The layer is created if it does not exist yet
l1 = ly.layer(1, 0)
```

Layouts and Cells

- Cells are identified in two ways
  - By an integer ID (“cell index”): does not hold a reference, can be checked for validity
  - By a db.Cell object
- Cells live in the Layout object and are managed by the latter
- Cells cannot be created as standalone objects
- Cell names should be unique

Layouts and Cells

```
# "ly" is a db.Layout object

# Gets the db.Cell object of
# the top cell
top = ly.top_cell()

# Gets the name of the top cell
print(top.name)

# Gets a cell by name or None
# if no such cell exists)
a = ly.cell("A")

# Gets the cell index of a cell
a_id = a.cell_index()

# Gets a Cell object from the ID
a_obj = ly.cell(a_id)
```

```
# "ly" is a db.Layout object

# Creates a new cell
# (chooses A$n name if A
# already exists)
a = ly.create_cell("A")

# renames the cell (Caution:
# may create duplicates)
a.name = "B"
```

The Cell object

- A Cell object contains
  - A list of instances (“placements”) of other cells
  - Correspondingly parent instances (where the cell is placed and how)
  - Child cells (the kind of cells placed)
  - Parent cells (which cells use this cell)
  - A shape container for each layer
  - A global bounding box and a per-layer bounding box

Cell bounding box

- The bounding box includes all child cells
- There is a global and a per-layer bounding box

```
# "ly" is a db.Layout object

top = ly.top_cell()

# Find layer 10/0
l10 = ly.layer(10, 0)

# The global bounding box in DBU units (a db.Box object)
bx_dbu = top.bbox()

# The global bounding box in µm units (a db.DBox object)
bx_um = top.dbbox()

# The bounding box of layer 10/0 in DBU units (a db.Box object)
bx_l10_dbu = top.bbox(l10)

# The bounding box of layer 10/0 in µm units (a db.DBox object)
bx_l10_um = top.dbbox(l10)
```

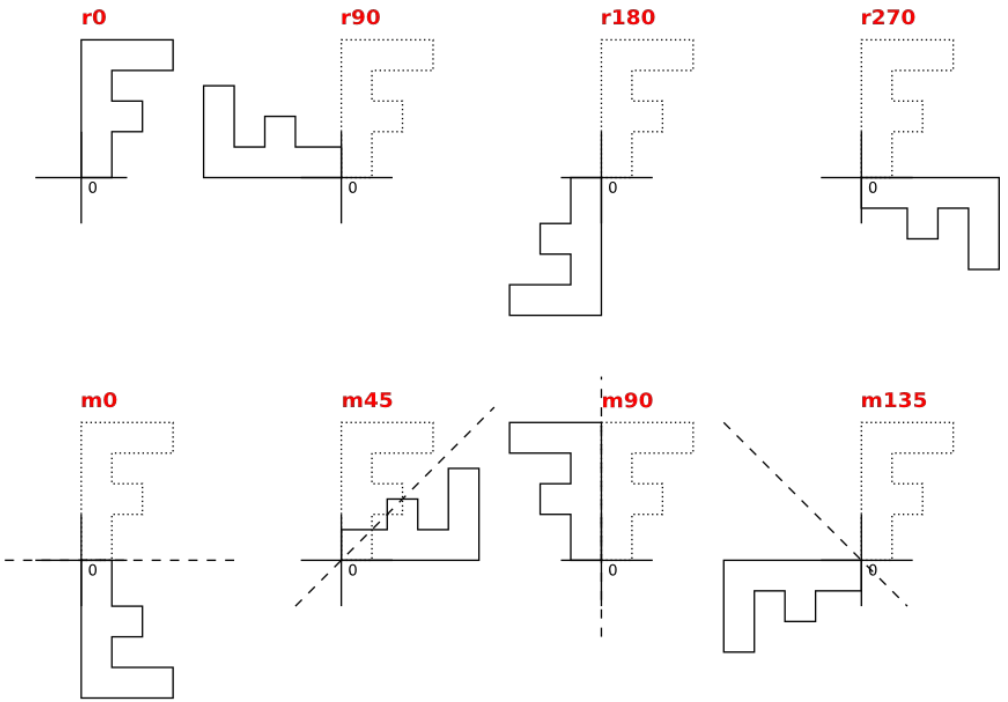
Transformations

- KLayout has a rich variety of transformation objects which convert a point into a differente space
  - Simple transformations (90 degree multiples, no scaling) used in standard layouts
  - Complex transformations (scaling, arbitrary rotation) for special layout styles
  - Complex transformations with type conversion
  - Matrix transformations (including perspective transformation) for use outside the geometry DB
- Resources: https://www.klayout.org/klayout-pypi/overview/transformations/

Simple transformations

- A Trans or DTrans object describes an special affine transformation with (in that order):
  - An optional mirror operation at the x axis
  - A rotation by 0, 90, 180 or 270°
  - A shift (translation) – Trans in DBU units, DTrans in µm units
- Transformations have
  - An inverse
  - A product ($T = T_1 * T_2$ is "T_1 applied after T_2")

8 basic fix-point operations



Creating simple transformations

```
import klayout.db as db

print(db.Trans.R0)
# -> r0 0,0

print(db.Trans(db.Trans.M90, 10, 20))
# -> m90 10,20

print(db.Trans(db.Trans.M90, db.Vector(10, 20)))
# -> m90 10,20

print(db.Trans(db.Trans.M90, 10, 20))
# -> m90 10,20

print(db.DTrans(db.DTrans.M45, db.DVector(0.4, -1.5)))
# -> m45 0.4,-1.5
```

Using simple transformations

```
import klayout.db as db

t1 = db.Trans.M45
t2 = db.Trans(db.Trans.R0, 10, 20)
print(t1 * t2)
# -> m45 20,10

print(t2.inverted())
# -> r0 -10,-20

print(db.Point(100, 200) * t1)
# -> 200,100

# Box arguments are: left, bottom, right, top
print(db.Box(10, 20, 15, 30)*t1)
# -> (20,10;30,15)
```

Applications of transformations

- Specify the placement of a cell in a cell instance
- Apply it to all kind of objects through the “*” operator
  - Boxes, Polygons, Paths, Labels, Points, Vectors, Edges, EdgePairs
  - CellInstArray
  - To specify placements in other contexts (Report database, Images etc.)

Complex transformations

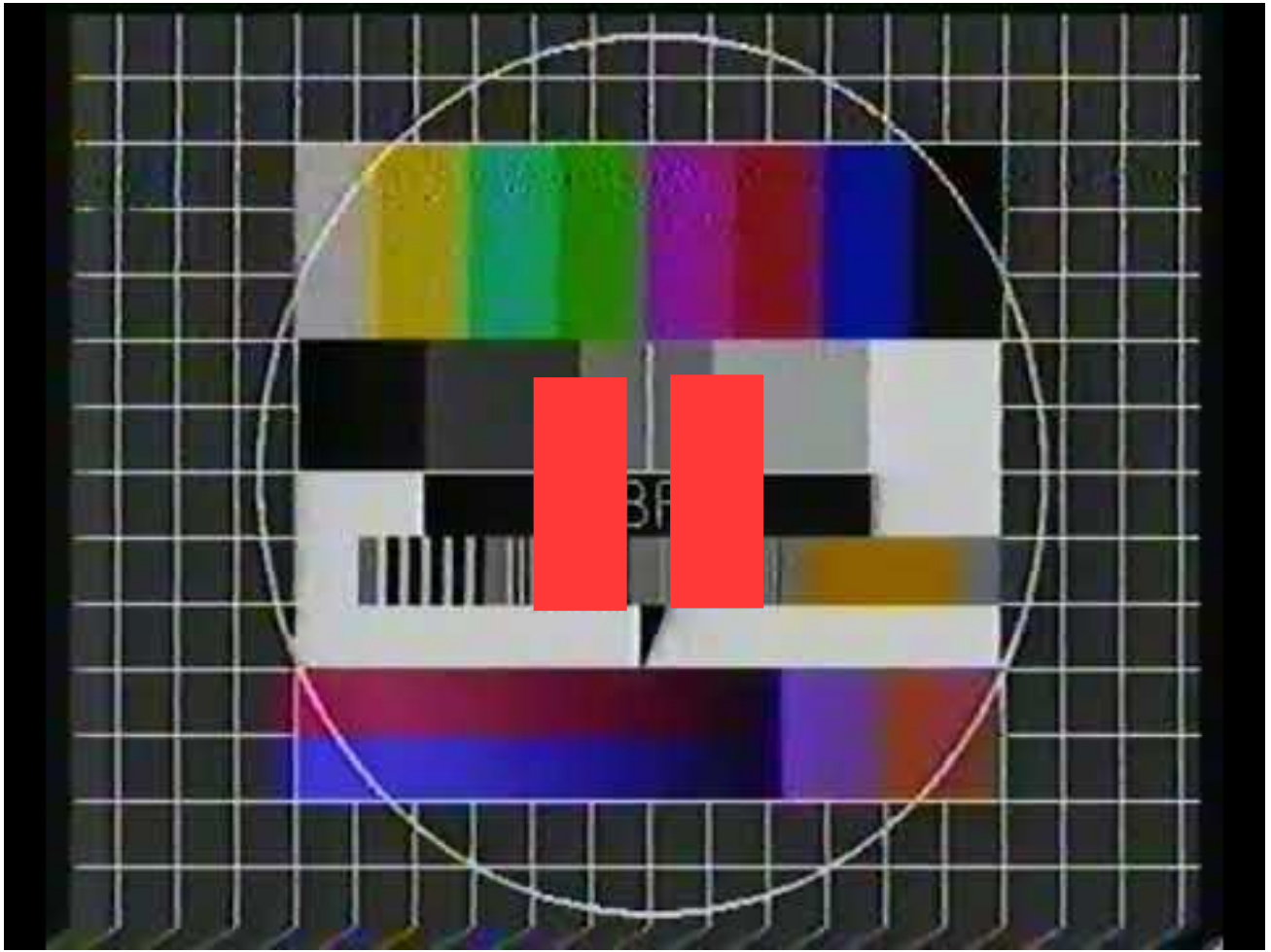
- Can have
  - Scale factor
  - Arbitrary rotation angle
  - Different input and output type
- Use in Layouts for cell instances is possible, but usually deprecated
  - Scaling devices is not a good idea
  - Arbitrary rotations imply grid issues and layout continuity problems
  - May be applicable for Photonics, MEMS or similar, but not MOS

Complex transformation types

Type	Input type	Output type	Comment
CplxTrans	Integer	Float	For DBU to µm conversion
VCplxTrans	Float	Integer	Inverse of CplxTrans
DCplxTrans	Float	Float	For use in Geometry DB API
ICplxTrans	Integer	Integer	

```
import klayout.db as db

dbu = 0.001
dbu2um = db.CplxTrans(dbu)
um2dbu = dbu2um.inverted()

print(dbu2um * db.Point(100, -200))
# -> 0.1,-0.2
print(um2dbu * db.DPoint(0.1, -0.2))
# -> 100,-200
```

Using CplxTrans and VCplxTrans to convert from DBU to µm and back



Let’s have some fun with the Python API!

https://github.com/klayoutmatthias/heichips2026_sokoban

Download the “sokoban.lym” file and place it in

- `c:\users\<you>\KLayout\pymacros` **on Windows**
- `~/.klayout/pymacros` **on Linux**

Restart KLayout and run in the menu:
“Macros/Examples/Sokoban”

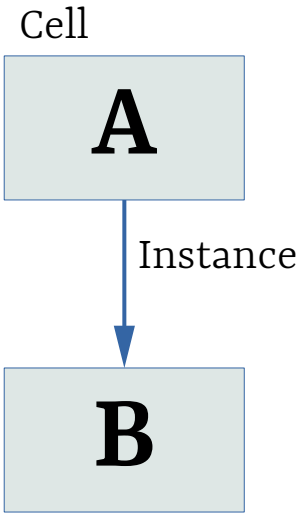
Ready to continue with the boring stuff?

Points and Vectors

- A Point or DPoint is an absolute position in 2d Cartesian space
- A Vector or DVector is a shift – the difference between two points
- Points and Vectors can be transformed: p’ = T*p and v’ = T*v
- Transformations act differently on Point or Vector
  - A vector does not see the translation while a point does
  - With this “$T*(p_2 – p_1)$” renders the same result when computing it separately: “$v = p_2 – p_1$” and “T*v”

Cells and instances

- Cells are placed inside other cells by creating an instance
- An instance has
  - a cell that it places
  - a transformation describing how the cell is placed
- Creating an instance of B in A makes
  - B a child cell of A
  - A a parent cell of B
- Cells and instances form a hierarchy graph
  - Cells as nodes, instances as edges



Single instances

- Use single instances to place a cell in another cell once
- A single instance is a special case of an array
- The working object is CellInstArray or DCellInstArray

```
import klayout.db as db

ly = db.Layout()
cell = ly.create_cell("TOP")

a = ly.create_cell("A")

# rotation 90 degree, displacement dx=100 DBU, dy=200 DBU
t = db.Trans(db.Trans.R90, 100, 200)

# place "A" with transformation t
cell.insert(db.CellInstArray(a, t))
```

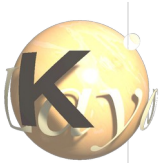
(D)CellInstArray API

- bbox(...): bounding box, also per-layer
- cell (set), cell_index (get,set): the cell placed
- is_regular_array(): True, if the instance is an array
- is_complex(): True, if the instance has a complex transformation
- a, b, na, nb (get,set): array parameters
- trans (get, set): basic simple transformation
- cplx_trans (get,set): basic complex transformation
- size(): number of cell placements in array
- each_trans()/each_cplx_trans(): all effective placements from the array

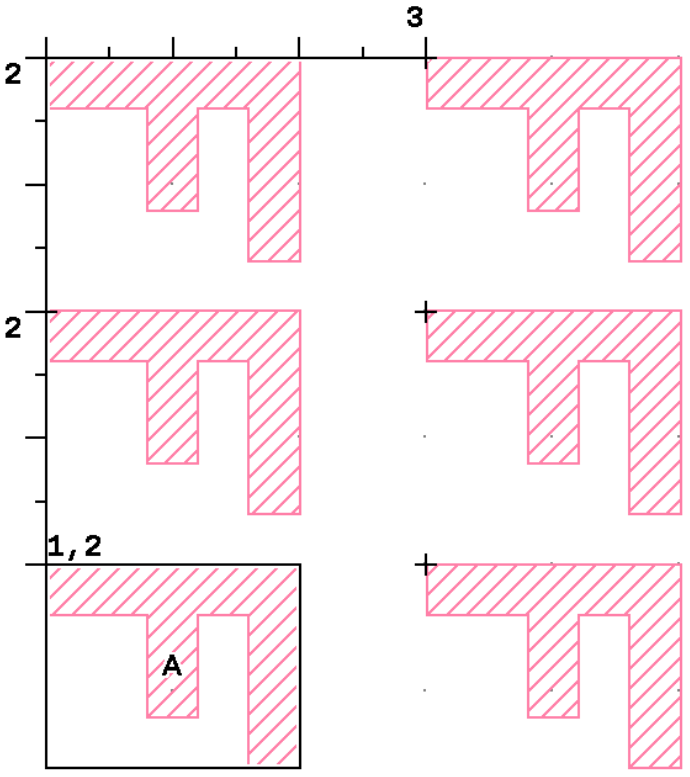
Instance arrays

- Instance arrays provide multiple displacements
  - $d_{i,j} = d_0 + i*a + j*b \quad (i = 0..n_a-1, j = 0..n_b-1)$

```
import klayout.db as db

ly = db.Layout()
cell = ly.create_cell("TOP")

a = ly.create_cell("A")

# rotation 270 degree, displacement dx=1.0µm, dy=2.0µm
t = db.DTrans(db.DTrans.R270, 1.0, 2.0)

# place "A" with transformation t and 2 columns, 3 rows
vc = db.DVector(3.0, 0)
vr = db.DVector(0, 2.0)
cell.insert(db.DCellInstArray(a, t, vc, vr, 2, 3))
```



Instance arrays



Position / Rotation / Scaling

Position of cell origin	x =	1.00000	y =	2.00000
Rotation angle (degree)		270	☐ Mirrored (at X-axis)	
Scaling factor (magnification)		1		

☑ Array Instance

This is instance [c,r] of array with

Columns =		2	Rows =	3
Column vector (x,y)	x =	3.00000	y =	0.00000
Row vector (x,y)	x =	0.00000	y =	2.00000

```
# rotation 270 degree, displacement dx=1.0µm, dy=2.0µm
t = db.DTrans(db.DTrans.R270, 1.0, 2.0)

# place "A" with transformation t and 2 columns, 3 rows
vc = db.DVector(3.0, 0)
vr = db.DVector(0, 2.0)
cell.insert(db.DCellInstArray(a, t, vc, vr, 2, 3))
```

The Instance object

- When you retrieve an instance from a cell, the reference is an Instance object

```
import klayout.db as db

# loop over all cell placements
for inst in cell.each_inst():

  # This is the Instance DB reference object
  print(inst)

  # Gets the CellInstArray working copy in DBU units
  print(inst.cell_inst)

  # Gets the DCellInstArray working copy in µm units
  print(inst.dcell_inst)
```

Instance API (selected)

- a, b, da, db, na, nb (get, set): get or change array parameters (“da”/”db” are µm versions of “a”/”b”)
- (d)trans, (d)cplx_trans (get, set): get or change the basic transformation
- (d)cell_inst (get, set): get or set (D)CellInstArray
- cell, cell_index (get, set): get of change the cell or ID of the cell that is placed
- pcell_parameter(), pcell_parameters(), pcell_declaration(), change_pcell_parameters(...): get or modify PCell parameters
- is_pcell(): True, if the instance is a PCell instance
- parent_cell (get, set): get parent cell, setting the parent cell will move the instance
- explode(): resolve array into single instance
- flatten(): replace instance by contents of the cell
- is_regular_array(), is_complex(): see CellInstArray

The “Shapes” shape container

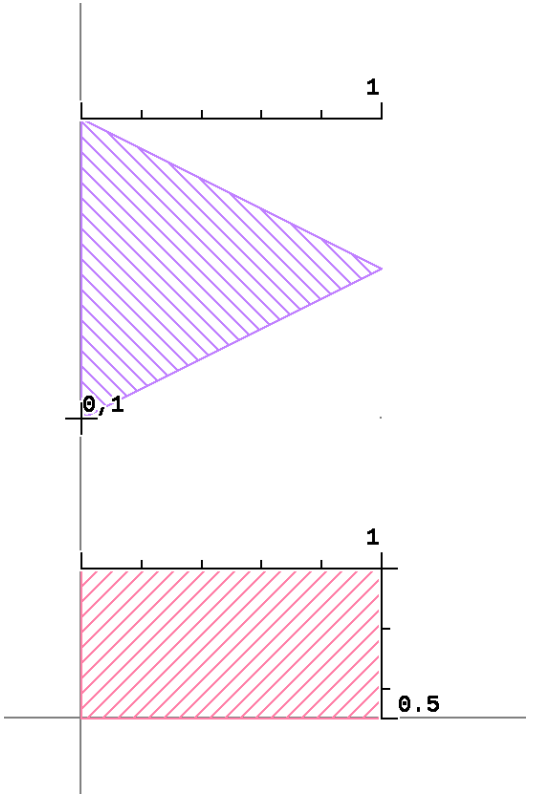
- A Cell has one shape container per layer
  - To access the Shapes container you need a layer ID
  - Use cell.shapes(layer_id) to get the Shapes container inside the cell on that layer
- The object for the shape container is “Shapes”
- The stored shapes can be boxes, polygons etc.
- Every shape can have properties – use (D)BoxWithProperties etc.
- The shapes are unordered
- A pointer into a “Shapes” container is a “Shape” object
- Shape containers can be used standalone

“Shapes” API

- Insert shapes, erase shapes, replace shapes
- Iterate shapes, with filter
- Region queries from rectangular region
- Reference to cell and layout the shape container is part of
- Transformations
- For operations, Region/Edge/EdgePairs/Texts are better containers

Creating shapes in Cell

```
import klayout.db as db

ly = db.Layout()
cell = ly.create_cell("TOP")

l1 = ly.layer(1, 0)
l2 = ly.layer(2, 0)

box = db.DBox(0, 0, 1.0, 0.5)
cell.shapes(l1).insert(box)

pts = [
  [ 0.0, 1.0 ],
  [ 0.0, 2.0 ],
  [ 1.0, 1.5 ]
]
polygon = db.DSimplePolygon(pts)
cell.shapes(l2).insert(polygon)
```



The Shape reference

- A shape inside a Shapes container is addressed by a Shape reference

```
shapes = db.Shapes()

# insert a box: shape is a reference
box = db.Box(0, 0, 10, 20)
shape = shapes.insert(box)
print(";".join([ str(s) for s in shapes.each()]))
# -> box (0,0;10,20)

# modify the shape inside the container
new_box = db.Box(-10, -20, 10, 20)
shape.box = new_box
print(";".join([ str(s) for s in shapes.each()]))
# -> box (-10,-20;10,20)

# remove the shape
shape.delete()
print(";".join([ str(s) for s in shapes.each()]))
# -> (empty)
```

Shape API

- Many methods!
- Conversion to type, set to type
- User properties
- Generic properties (area, perimeter, bbox ...)
- Transformation
- Delete shape
- Edge iterators, point iterators
- Reference to layer, shape container
- ...

Caution with iterate+modify

- Iterators may behave weird if you delete or modify shapes while iterating

```
# shapes is a db.Shapes container

# DO NOT DO THIS
for s in shapes.each():
  if s.area() < 1000:
    s.delete()

# instead:
to_del = [ s in shapes.each() if s.area() < 1000]
for s in to_del:
  s.delete()
```

(D)Box

- A box has four coordinates: left, bottom, right, top
- A box can be empty (a null box)
  - This is useful to describe the bounding box of an empty layer
- Methods exist for computing
  - Box unions (box enclosing two boxes)
  - Box intersections
  - Tests (touching, overlapping, inside)
  - Enlarged/shrinked boxes

Box API

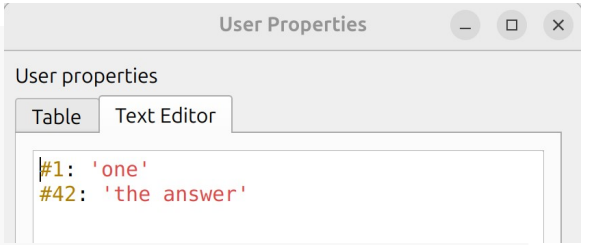
```
import klayout.db as db

bx = db.Box(10, 20, 30, 50)
print(bx)
# -> (10,20;30,50)
print(bx.width())
# -> 20
print(bx.height())
# -> 30

bx = bx.enlarged(1, 2)
print(bx)
# -> (9,18;31,52)
```

```
import klayout.db as db

bx1 = db.Box(10, 20, 30, 50)
bx2 = db.Box(15, 25, 35, 55)

# intersection
print(bx1 & bx2)
# -> (15,25;30,50)

# enclosing box
print(bx1 + bx2)
# -> (10,20;35,55)
```

(D)BoxWithProperties

- Creates a box with properties attached

```
import klayout.db as db

properties = {
  1: "one",
  42: "the answer"
}

bx = db.BoxWithProperties(db.Box(10, 20, 30, 50), properties)

print(bx)
# -> (10,20;30,50) props={#1=>'one',#42=>'the answer'}
```



(D)Polygon, (D)SimplePolygon

- A simple polygon is a piecewise linear graph made from N points forming a loop (the “hull”)
- A (generic) polygon can have holes too
- The loops (“contours”) are oriented automatically to be clockwise for the hull and counterclockwise for the holes
  - This way, the “inside” of the polygon is always “right” when walking around a loop
- Only simple polygons are read from and written to GDS or OASIS
- Generic polygons are converted to simple ones when writing

Simple polygons

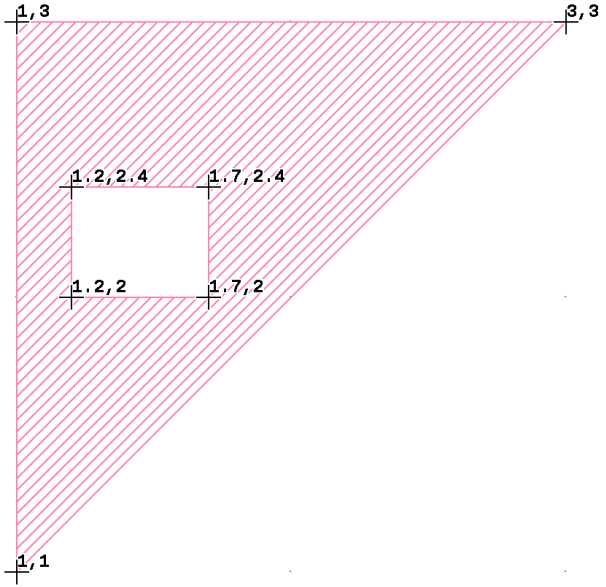
- How to create a simple polygon

```
import klayout.db as db

# can be array of db.DPoint or [x,y] pairs
pts = [ [1.0,1.0], [1.0,3.0], [3.0,3.0] ]

polygon = db.DSimplePolygon(pts)

print(polygon)
# -> (1,1;1,3;3,3)
```

Generic polygons

- How to create a polygon with a hole

```
import klayout.db as db

# can be array of db.DPoint or [x,y] pairs
pts = [ [1.0,1.0], [1.0,3.0], [3.0,3.0] ]
hole_pts = [ [1.2,2.0], [1.2,2.4],
             [1.7,2.4], [1.7,2.0] ]

polygon = db.Dpolygon(pts)
polygon.insert_hole(hole_pts)

print(polygon)
# ->
(1,1;1,3;3,3/1.2,2;1.7,2;1.7,2.4;1.2,2.4)
```

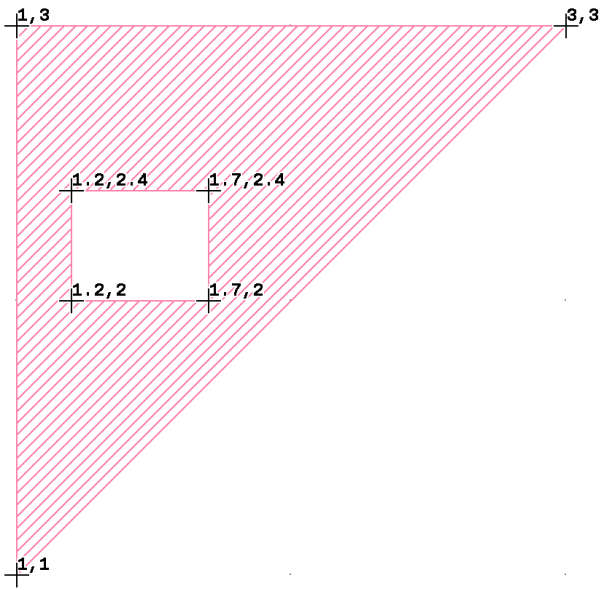


Simple and generic polygon API

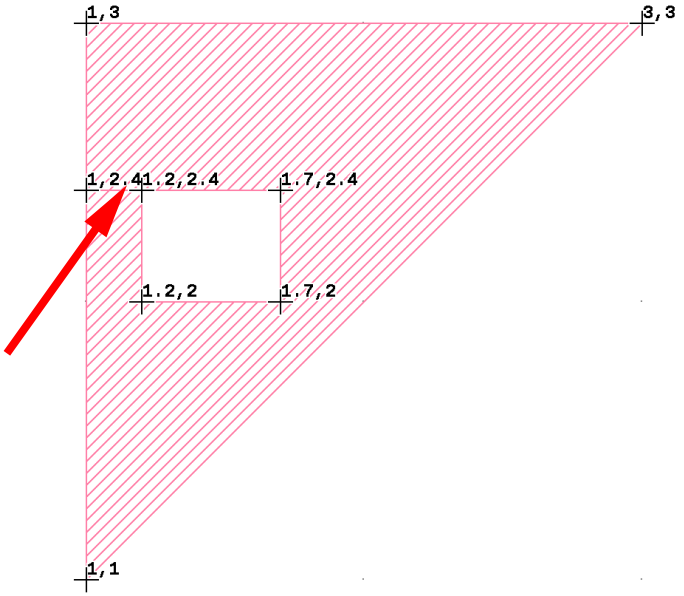
- area(), perimeter(): gets area and perimeter
- break(…): break into smaller parts
- decompose_convex(), decompose_trapezoids(), delaunay(), hm_decomposition(): decomposition
- bbox(): gets the bounding box
- each_edge(), each_point(): Iterate edges, points
- is_convex(), is_empty(), is_rectilinear(), is_halfmanhattan(): predicates
- minkowsky_sum(): apply a “pen” to the contour

Resolving holes

- Use “to_simple_polygon()” (only available for Polygon, not for DPolygon)



Polygon



After to_simple_polygon():
a single loop

More algorithms on polygons

- The “Region” class is basically a collection of polygons and offers a lot of additional functionality for polygons
  - Booleans
  - Merge
  - Sizing
  - Interaction tests
  - Decomposition
  - …

(D)Edge

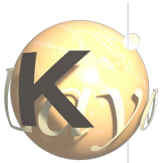
- The edge object is not a shape you use in layouts, but is used for example to iterate the edges of a polygon or inside DRC scripts
- It is a directed line connecting two points: p1 and p2
- The (D)Edge API has methods to
  - check intersections
  - test which side of the edge a point is
  - compute distances
  - clip edges
  - ...

(D)EdgePair

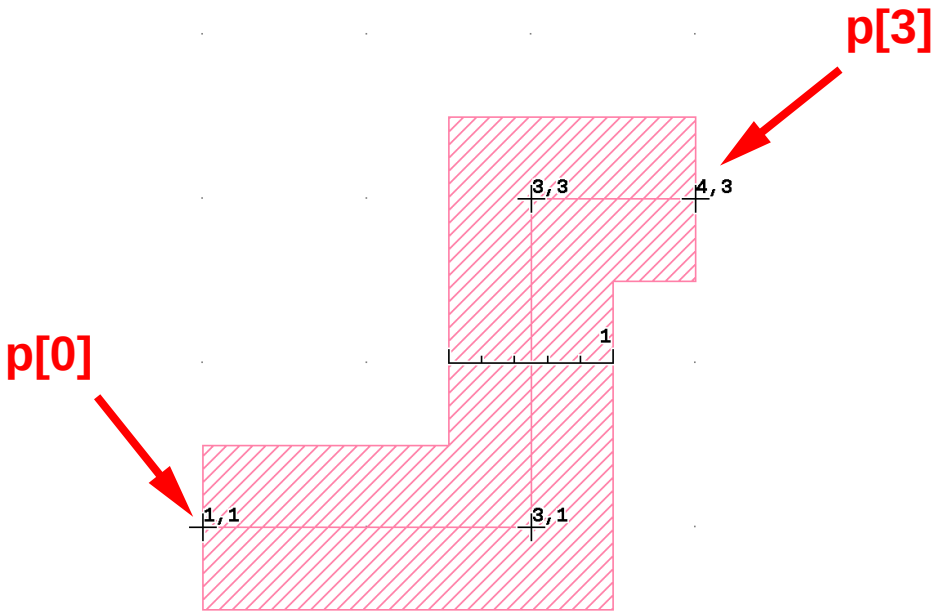
- The edge object is not a shape you use in layouts, but is used for example to iterate the edges of a polygon or inside DRC scripts
- It consists of two edges: first and second
- It is used in DRC functions to represent the two edges of a distance/width/overlap violation marker

(D)Path

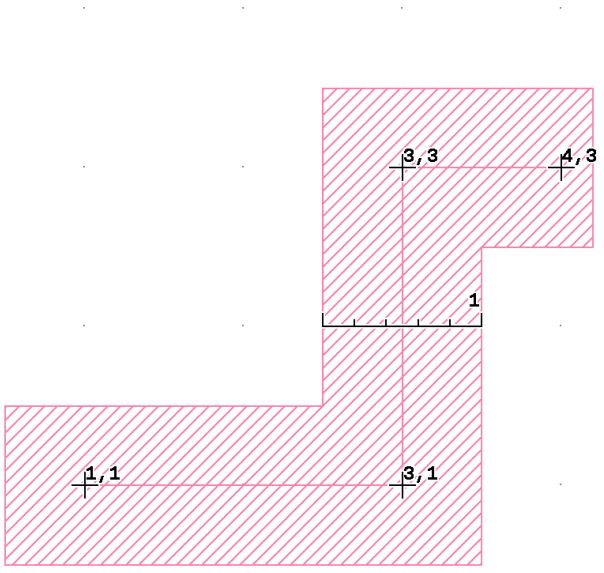
- A path is a line with a certain width
- The ends can be flush or have a specific extension
- Round ends are a deprecated specialty of GDS
- Beware of odd widths (in DBU units)
- Beware of short segments



Path width and extensions

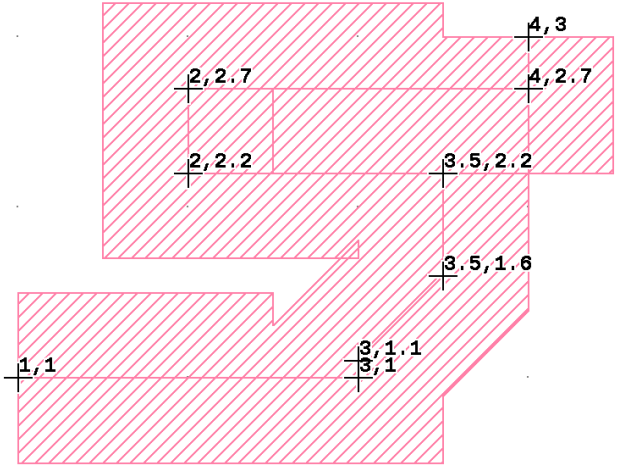


A path with four points, width = 1
and flush ends (bgnext = endext = 0)



The same path with variable
extensions: bgnext = 0.5, endext = 0.2

Pathologic paths



In general, for compatibility with other tools, avoid:

- segments that are shorter than the path's width (half the path width for first and last segment)
- Non-manhattan segments
- Acute angles
- Single-point paths
- Off path widths (in DBU units)

Path API

- How to create a path

```
import klayout.db as db

# can be array of db.DPoint or [x,y] pairs
pts = [ [1.0,1.0], [3.0,1.0], [3.0,3.0], [4.0,3.0] ]

Width = 1.0
bgnext = 0.5
endext = 0.2
path = db.DPath(pts, width, bgnext, endext)

print(path)
# -> (1,1;3,1;3,3;4,3) w=1 bx=0.5 ex=0.2 r=false
```

Path API

- each_point(): Iterator over every point
- bbox(): gets the bounding box (approximation)
- bgn_ext, end_ext, width (get, set)
- area(), perimeter(): gets the area and perimeter of the path (approximation)
- length(): gets the length of the path
- Polygon(), simple_polygon(): converts the path to a polygon or simple polygon
- ...

(D)Text

- A text is a location (a point) with a string (also called “label”)
- Optionally, it has an orientation, a height and justification attributes
- Only point an label are written to and read from OASIS files
- Texts are used to label a position or placed over a shape to attach a label to it (e.g. a net name)
- Currently, there are no individual fonts, but one text font can be configured in the application for all labels in a layout
- The area and bounding box of a text is a point only and does not include the text area

Text API

- How to create a text

```
import klayout.db as db

# a text with a position
text = db.DText("ABC", db.DTrans.R0, 1.0, 2.0)
print(text)
# -> ('ABC',r0 1,2)

# a text with an orientation and character size
text = db.DText("ABC", db.DTrans(db.DTrans.R90, 1.0, 2.0))
Text.size = 0.5
print(text)
# -> ('ABC',r90 1,2) s=0.5
```

Text API

- halign, valign (get, set): text alignment options
- bbox(): gets the bounding box (a single point)
- x, y (get, set): the location
- trans (get, set): the location and rotation packed into a (D)Trans object

Recursive queries

- The RecursiveShapeIterator and RecursiveInstanceIterator objects allow iterating shapes or instances of a cell “as if flat”
  - i.e. it will descend into child cells
- During iteration it delivers
  - A Shape or Instance for the current object
  - The cell the current object is in
  - For shapes, the layer the object is on
  - The transformation indicating how this object appears in the initial cell

RecursiveShapeIterator API

- The query can be confined to a complex or single-box region in the initial cell
- The iterator can deliver shapes from single or multiple layers
- The type of shapes addressed can be configured
- The maximum iteration depth can be configured
- The instance path for the current object is available
- The cell tree to be iterated can be customized by enabling or disabling specific cells

RecursiveShapeIterator API

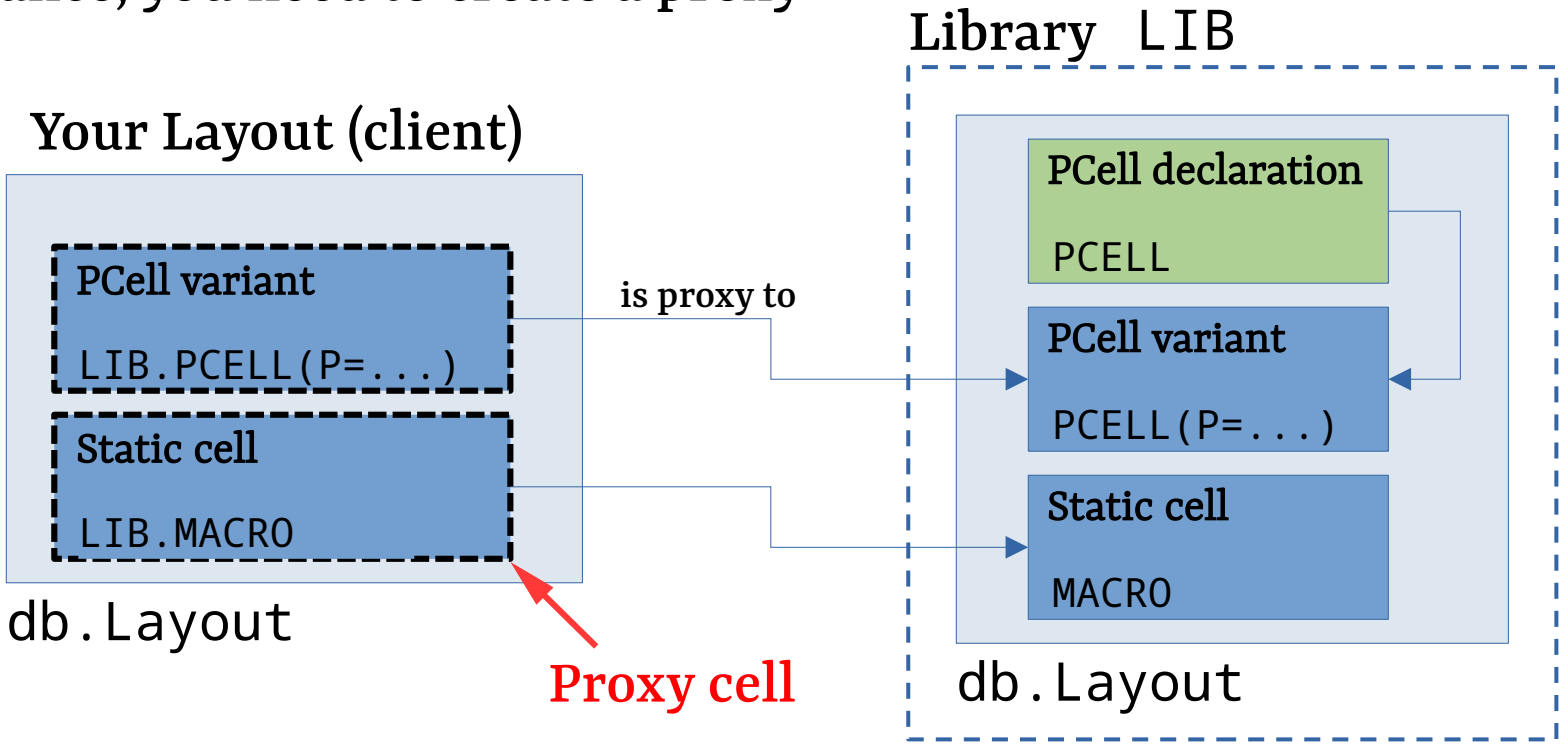
```
# cell is a db.Cell
# li is a layer ID

# Reports all shapes from layer li which are "polygon-like"
# by converting to polygons and as seen in cell:

it = cell.begin_shapes_rec(li)
for i in it.each():
  poly = i.shape().polygon
  if poly is not None:
    print(i.trans() * poly)
```

RecursiveInstanceIterator API

- The query can be confined to a complex or single-box region in the initial cell
- The current object is an Instance
- It will also visit the instance that are further iterated
- The iteration can be confined to instances of specific cells (“targets”)
- The maximum iteration depth can be configured
- The instance path for the current object is available
- The cell tree to be iterated can be customized by enabling or disabling specific cells

PCell and Library instances

In order to reference a cell from a library and/or to create a PCell instance, you need to create a proxy

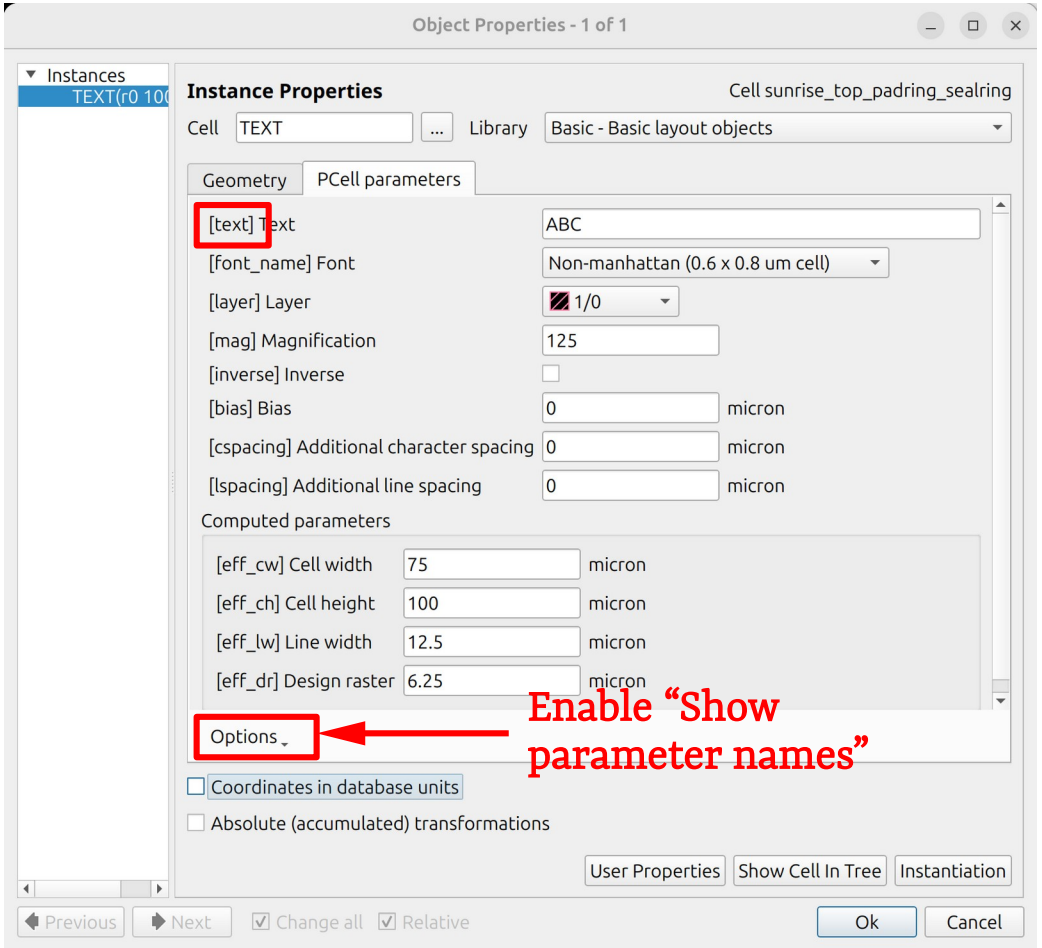


PCell and Library instances

- This creates a PCell proxy

```
# ly is a db.Layout
# cell is a db.Cell

cell_name = "TEXT"
lib_name = "Basic"
param = {
  "text": "ABC",
  "mag": 17.5,
  "layer": db.LayerInfo(1, 0)
}

proxy = ly.create_cell(cell_name, lib_name, param)

cell.insert(db.DCellInstArray(proxy, db.DVector(100, 200)))
```

PCell parameter names



You can look up the parameter names in the PCell instance properties page

User Properties

- Layout, Cell, Shape and Instance can have user properties attached
  - User properties are key/value pairs
  - Only Shape and Instance can have user properties in GDS2 and keys need to be integers while values need to be strings with limited length (formally <32768)
- The following functions manage properties:
  - property, properties, set_property, set_properties, clear_properties, delete_property
- Internally a property set is identified by a global unique ID
- For the working objects with properties use BoxWithProperties etc.

Pitfall: iterate / manipulate

- When you iterate shapes or instances it is often not safe to manipulate them while iterating
  - When the DB reorganizes due to changes, the iterators may become disturbed
- Safe way is to collect Shape/Instance objects first and manipulate or remove them in a second pass

The Region object

- A Region is a collection of polygons
- The order is undefined
- It can be created from a layout cell (original region) or by populating an empty Region object with “polygon-like” objects
- It has a huge number of methods
- It is the basis of the DRC feature
- It is only available in integer unit space
- Regions are the #1 choice when it comes to polygon processing

Standalone Region objects

- Computes the boolean NOT of two polygons:

```
import klayout.db as db

r1 = db.Region()
r1.insert(db.Box(0, 0, 1000, 1000))

r2 = db.Region()
r2.insert(db.Box(100, 200, 1000, 1000))

r1not2 = r1 - r2
print(str(r1not2))
# -> (0,0;0,1000;100,1000;100,200;1000,200;1000,0)
```

Original layer Regions

- Are created from a RecursiveShapeIterator
- Can be bound to a hierarchical engine for hierarchical operations
- A cheap unless you derive other Regions from them

More containers

- Edges: a collection of Edge objects
  - Is the basis of edge layers in DRC
- EdgePairs: a collection of EdgePair objects
  - Is the basis of edge pair layers (e.g. results from DRC checks)
- Texts: a collection of Text objects

More from the db module

- Netlist API: a number of classes to represent hierarchical netlists
  - Circuit, devices, pins, nets, subcircuits ...
  - With I/O (SPICE)
  - Supports netlist manipulation
- LayoutToNetlist
  - Extraction of connectivity and devices from a layout
  - Includes geometries for nets
- LayoutVsSchematic
  - The LVS database
  - Includes LayoutToNetlist
  - Includes Schematic
  - Includes a cross-reference

The Application API

Application Singleton

- Provides application-wide services
  - Manages the MainWindow object
  - Manages the central configuration DB

MainWindow object

- Represents the main application window
  - Is a Qt object if Qt is enabled
  - Manages the LayoutViews

LayoutView object

- Represents one tab in the central window
  - Manages the layouts loaded (“CellView”s)
  - Manages the layer views and styles
  - Manages markers (highlights)
  - Manages rulers (aka “annotations”) and images
  - Manages marker DBs and Netlist DBs loaded along with the layouts
  - Manages display options (background, hierarchy levels, grid etc.)
  - Provides hooks and an API for plugging in new tools
  - Manages undo/redo

LayoutView plugins

- Plugins allow implementing new tools and place them in the toolbar
  - Tools get “activated” when selected
  - During active state, they receive mouse and key events
  - It is possible to associate tool option pages and overlay views that become active with the plugin
  - They allow implementing all kind of new features
  - Example: “Align tool” by Martin Köhler / JKU
    https://github.com/iic-jku/klayout-align-tool

Not Covered Here ...

Ask your favorite AI for details …

- Qt bindings
- In-depth discussion of the application API
- Algorithm section of the db module (like R extraction, decomposition etc.)
- Hundreds of DRC features baked into the API
- I/O
- TilingProcessor
- Introspection
- Documentation generators
- LayoutQuery
- Expressions
- …

Thank you for Listening