

# BEAMER & Python

- BEAMER allows for
  - Multiple operations targeting **layout modification**
  - **Proximity corrections** (PEC)
  - Setting complex **exposure** strategies



- **Python** allows for
  - Challenging tasks performed in few steps
  - Editing and running code easily



**Python** with **BEAMERpy** is a great tool for automation

# BEAMERpy

- **Python** is a modern programming language that allows designing **BEAMER** process flows from small scripts up to full scale applications.
- **Python** is a post processing language where commands are written down, executed and successively processed. There is no compiling involved making this a very flexible language.
- **BEAMER** integrates itself to **Python** by providing a library and allowing to execute operations as they do in the graphical flow.

Flow can run in **Python** IDEs. For this, **Python** has a basic structure:

1. Header which includes system settings and the **BEAMER** library

The first two load system libraries

```
# load necessary system libraries  
import os  
import sys
```

Next lines set the paths where **Python** finds the libraries from **BEAMER** and then imports them

```
# set the path to the BEAMER python interface libraries  
sys.path.append("D:/Work/bin/BEAMER/v7.1.0_x64")  
import BEAMERpy
```

2. Python calls functions from a library using the structure:

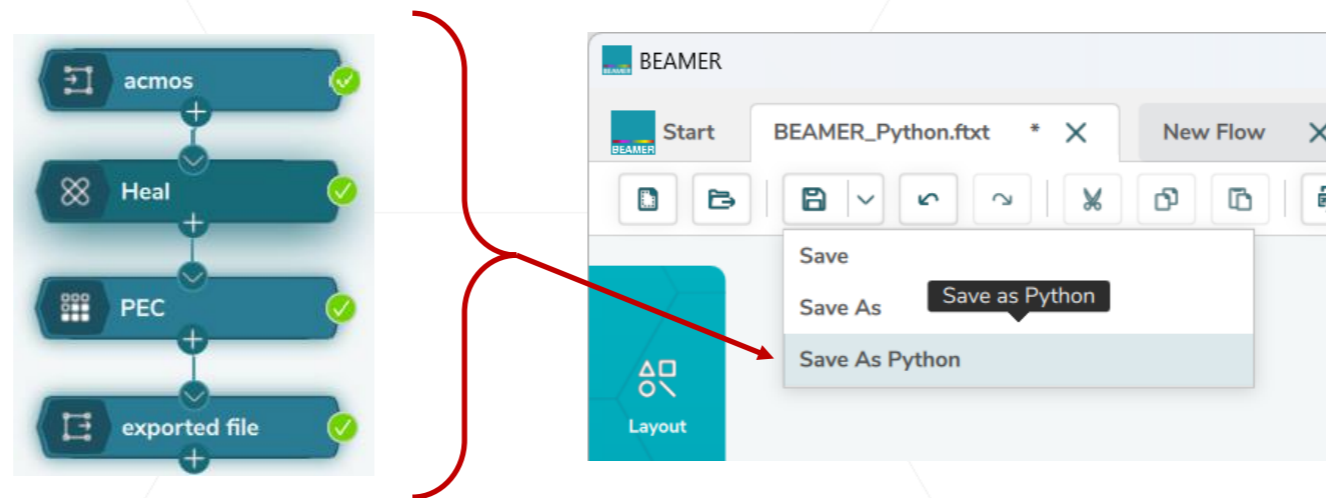
<name of libraries>.<function in library>

We abbreviate this a bit

*# load the python BEAMER object*  
`BEAMER = BEAMERpy.GBEAMER()`

Now can begin to program our **BEAMER** actions in **Python**. For this we need to know how to get the proper commands

3. All modules in **BEAMER** have settings which are entered in the **Python** code. The easiest way to get to the full commands is **saving** a flow **as Python**



This will export your current flow as a **Python** script (\*.py). Use an editor as Notepad to open and read this information

The exported flow will be a *ready to run* script



Only specified parameters affect the module behaviour. Parameters not included use **Default** values

```
# Python file exported with BEAMER Revision Number 7.1.0 (84289), Feb
26 2024
# exported at 2024-Apr-19 19:23:40
```

```
# load necessary system libraries
import os
import sys

# set the path to the BEAMER python interface libraries
sys.path.append("D:/Work/bin/BEAMER/v7.1.0_x64")
import BEAMERpy

# load the python BEAMER object
BEAMER = BEAMERpy.GBEAMER()

# set the PSF Archive Directories
BEAMER.set_psf_archive_folder({ 'ArchivePath' :
'C:/Users/Public/Documents/.GenISys/TRACER/2D.Archive' })
```

Header

```
gobj_1 = BEAMER.import_gds( { 'FileName' :
'D:/Work/examples/BEAMER/Layouts/acmos.gds' } )
```

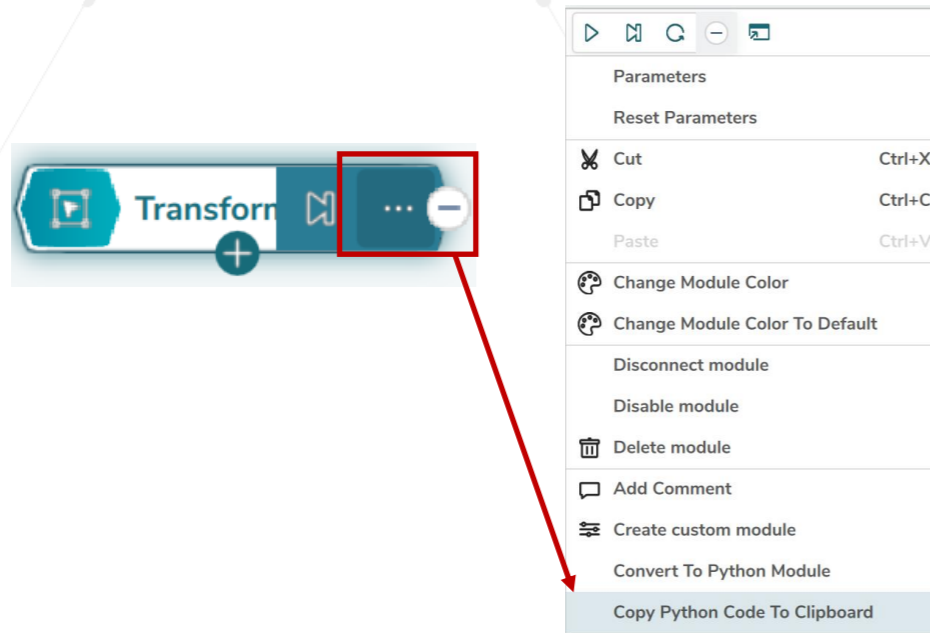
Modules

```
gobj_2 = BEAMER.heal( gobj_1, { 'TargetLayer' : '1(0)' } )
```

```
gobj_3 = BEAMER.pec( gobj_2, { 'PSFArchiveIdentifierString' :
'Substrate_Si_Thickness_700000_Energy_100_Layers__Resist_PMMA_100_nm_Z-
Position_0.045_Electrons_2000000_Alpha_0_Beta_0_Eta_0_Gamma1_0_Nue1_0_G
amma2_0_Nue2_0_Simulator_mcTrace_1.0.0' } )
```

```
gobj_4 = BEAMER.export_gds( gobj_3, { 'FileName' :
'D:/Work/examples/BEAMER/Layouts/acmos_pec_exported.gds' } )
```

- Commands for single modules can be copied to clipboard:



```
transform(gobj_1 ,
          {'CoordinateOrigin' : 'LayoutOrigin',
           'CenterMode' : 'XY',
           'ScaleX' : 1.000000,
           'ScaleY' : 1.000000,
           'ShiftX' : 0.000000,
           'ShiftY' : 0.000000,
           'Rotation' : 0.000000,
           'ReflectX' : False,
           'ReflectY' : False} )
```

**IMPORTANT:** With this method, only the command is copied. User needs to add **BEAMER** library!



```
gobj_2 = BEAMER.transform(gobj_1 ,
                          {'CoordinateOrigin' : 'LayoutOrigin',
                           'CenterMode' : 'XY',
```

Python allows parameters. These can be a file name or a size of the field or other variables.

The example here reads the first parameter as the name of the file to be imported: *varprefix*

The second argument defines the export format: *varformat*

Note: The file is exported in the same folder as the .py script

```
varprefix = "processed_"
varformat = ".gds"

# create new GLayoutBeamer Object
beamer = BEAMERpy.GBEAMER()

# load layout; elements in layout are on layer 1
layout = beamer.import_gds( { 'FileName' : sys.argv[1]} )

# save result as gds-ii
if varformat == ".gds":
    beamer.export_gds(layout, { 'FileName' : varprefix +
sys.argv[2] + ".gds" } )
elif varformat == ".ldb":
    beamer.export_ldb(layout, { 'FileName' : varprefix +
sys.argv[2] + ".ldb" } )
elif varformat == ".cif":
    beamer.export_cif(layout, { 'FileName' : varprefix +
sys.argv[2] + ".cif" } )
else:
    beamer.export_txl(layout, { 'FileName' : varprefix +
sys.argv[2] + ".txl" } )
```

- `sys` and `os` modules allow interaction with the operative system and calling external libraries.

General structure to add libraries and call external libraries

```
import os  
import sys
```

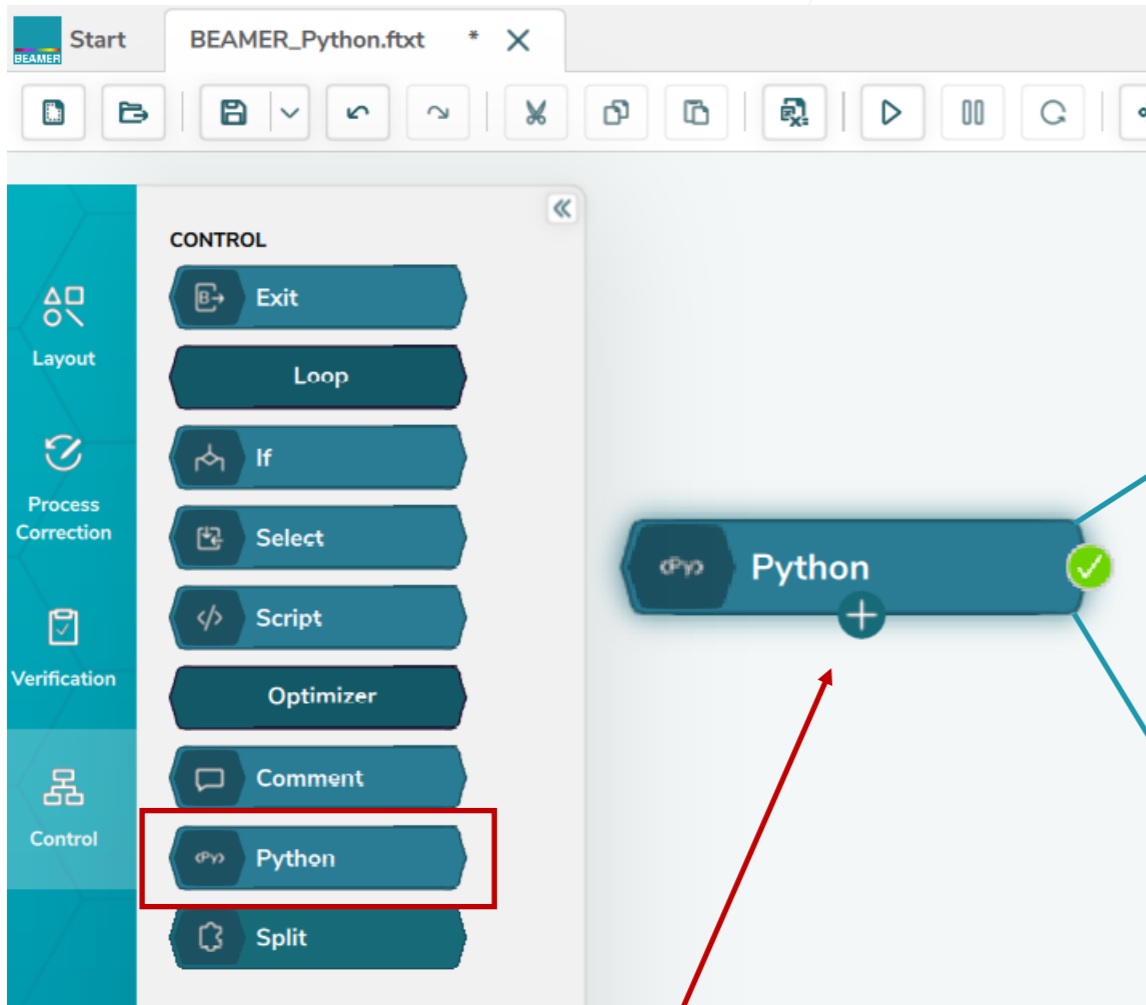
```
sys.path.append("D:/Work/bin/BEAMER/v7.1.0_x64")
```

```
import BEAMERpy
```

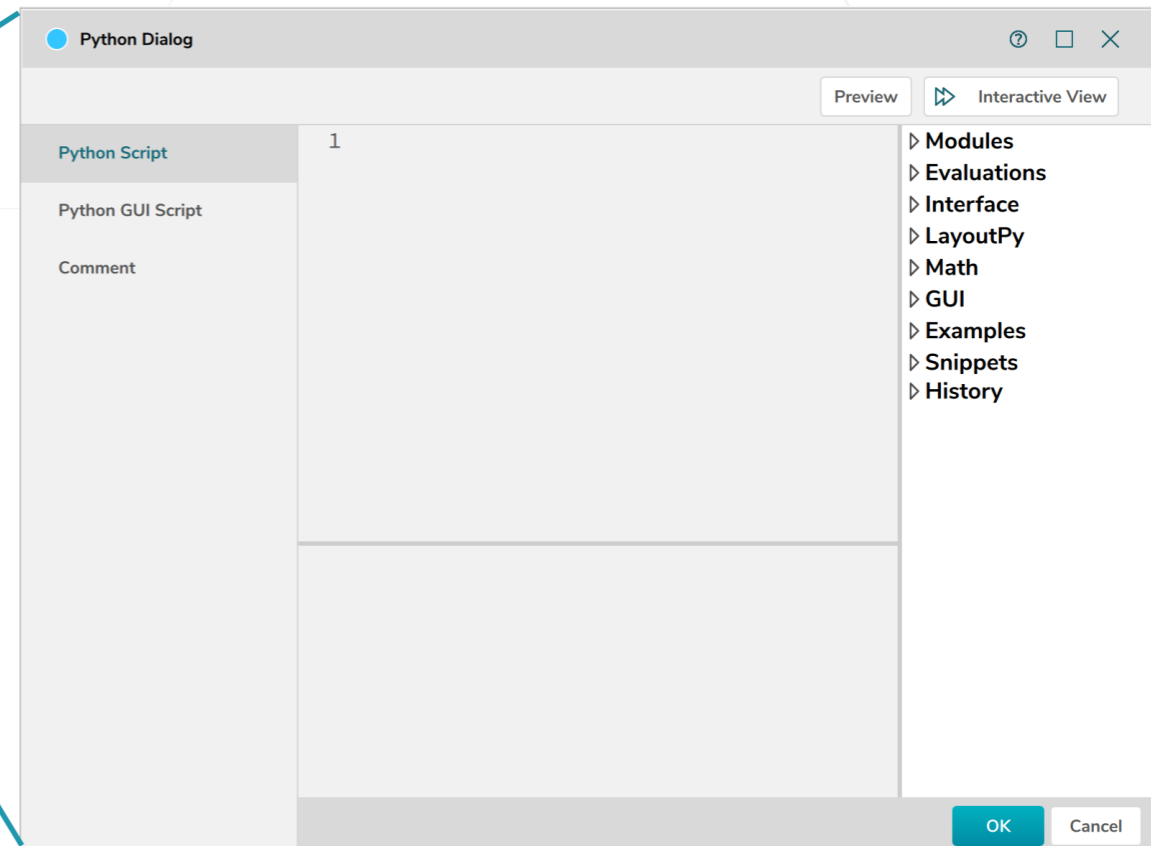
← Library in the path defined

System libraries are by default installed in Windows and Linux. If these are not:  
<https://www.geeksforgeeks.org/how-to-install-os-sys-module-in-python/>

# Python Module



BEAMER provides a **Python module** (under the Control modules) that serves as a scripting interface (API)



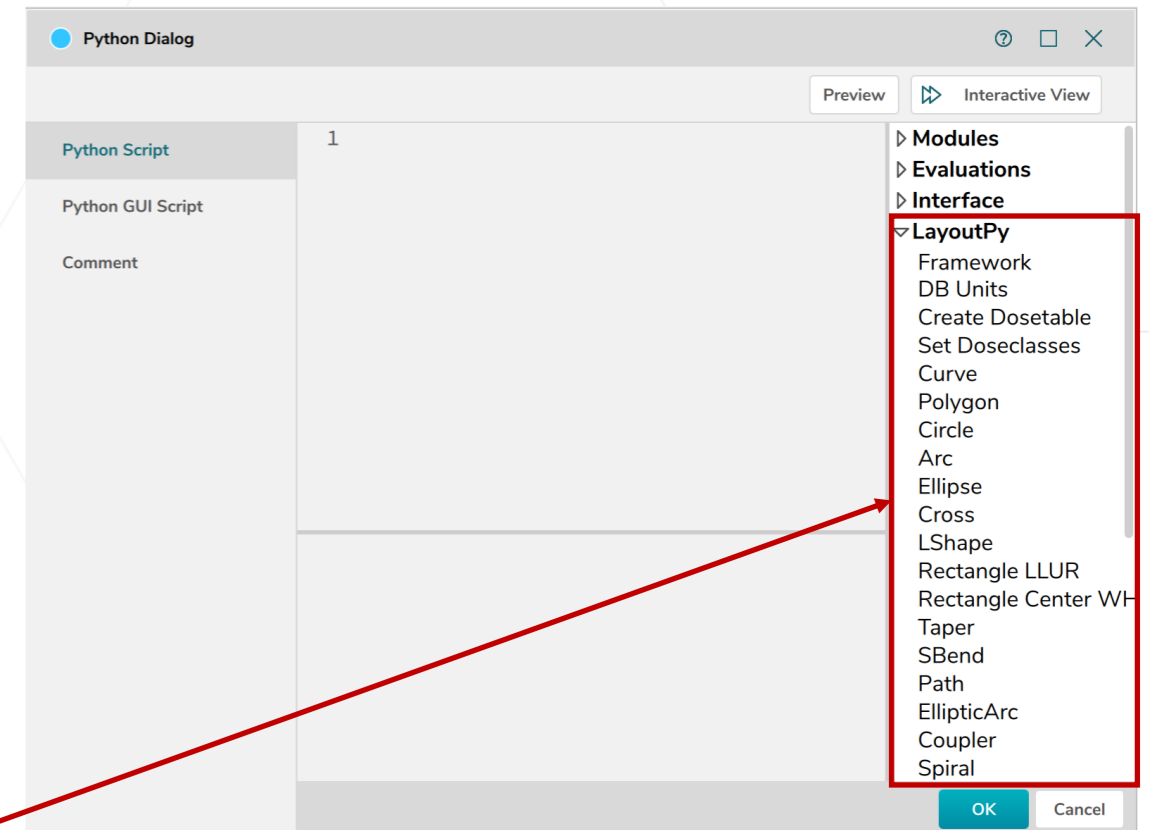
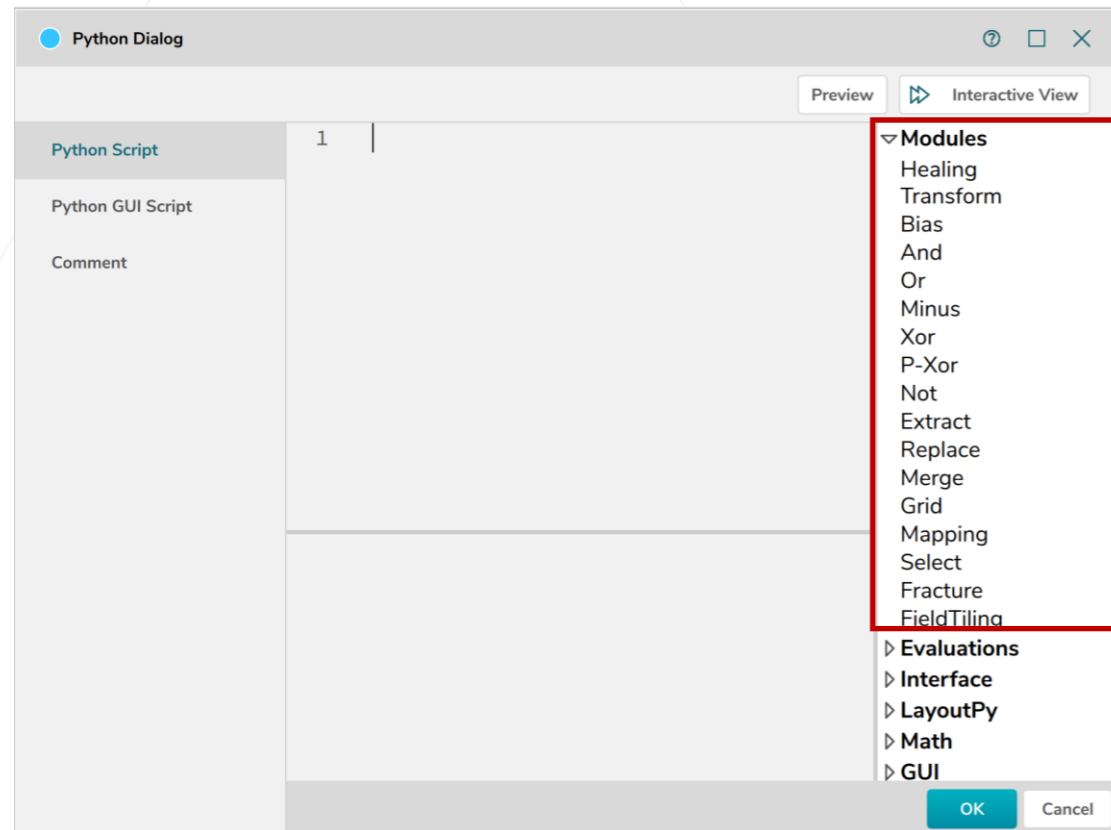
Drag and drop the module to the working space

BEAMER  
modules



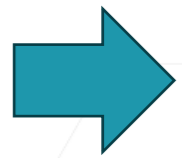
Python  
scripting

LAYOUTPy  
design



Drag and drop the commands to the scripting area or double click them

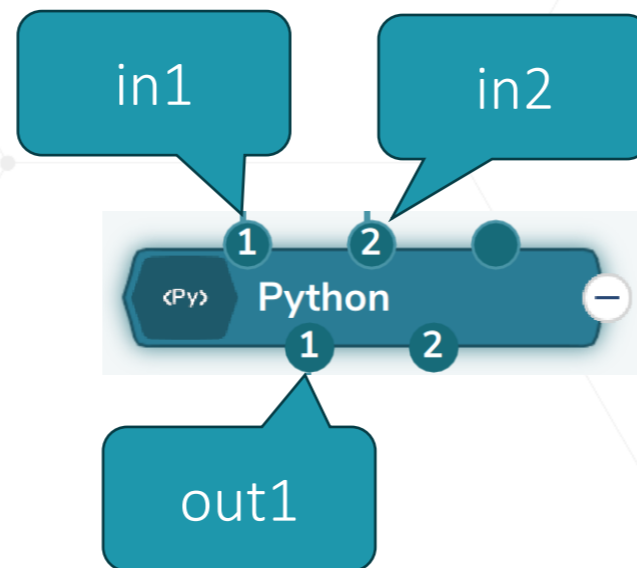
- The **Python** module uses two keywords for handling layouts



**in** + number (in1, in2...)

**out** + number (out1, out2...)

The ports are viewed from left to right and are numbered in that order



Output ports appear as they are created within the script

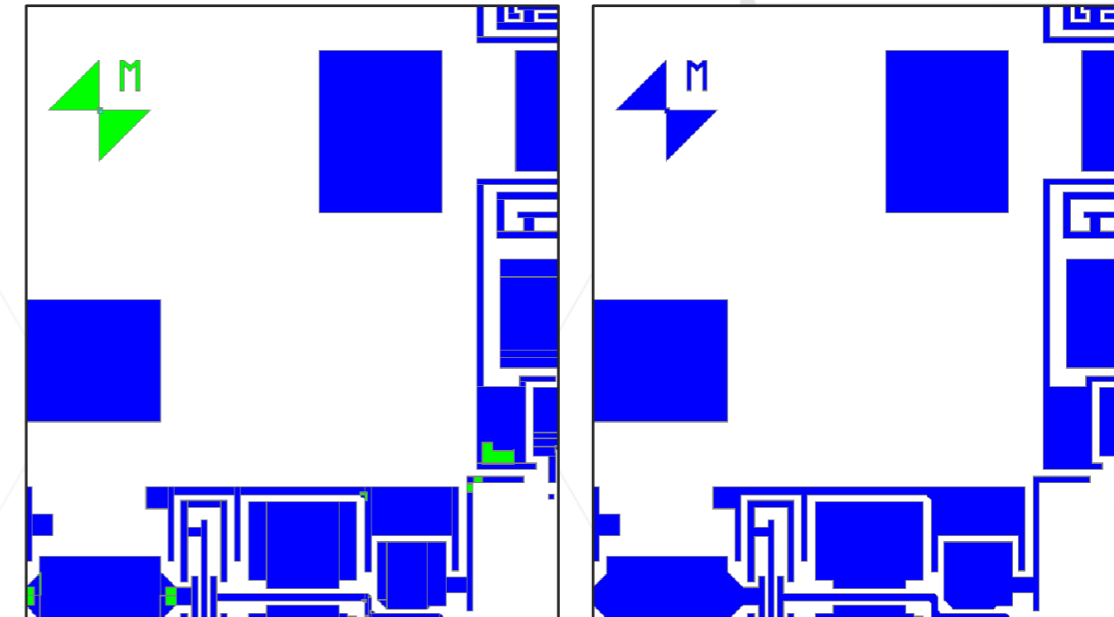
# Modifying Layouts via Python

1. Import a layout into **BEAMER**
2. Connect a **Python** module
3. Make modifications to the layout using **Python** (drag and drop functions from right-panel)
4. Press **Preview** to run the flow within the module environment



Imported

Healed



```

1 healing = BEAMER.heal(in1,
2   {'TargetLayer' : '1(0)',
3   'SoftFrame' : 0.300000,
4   'HierarchicalProcessing' : True,
5   'LayerAssignment' : 'AllLayer',
6   'ProcessingMode' : 'Healing'})
7 out1 = healing
  
```

Interactive View

Modules  
Healing  
Transform

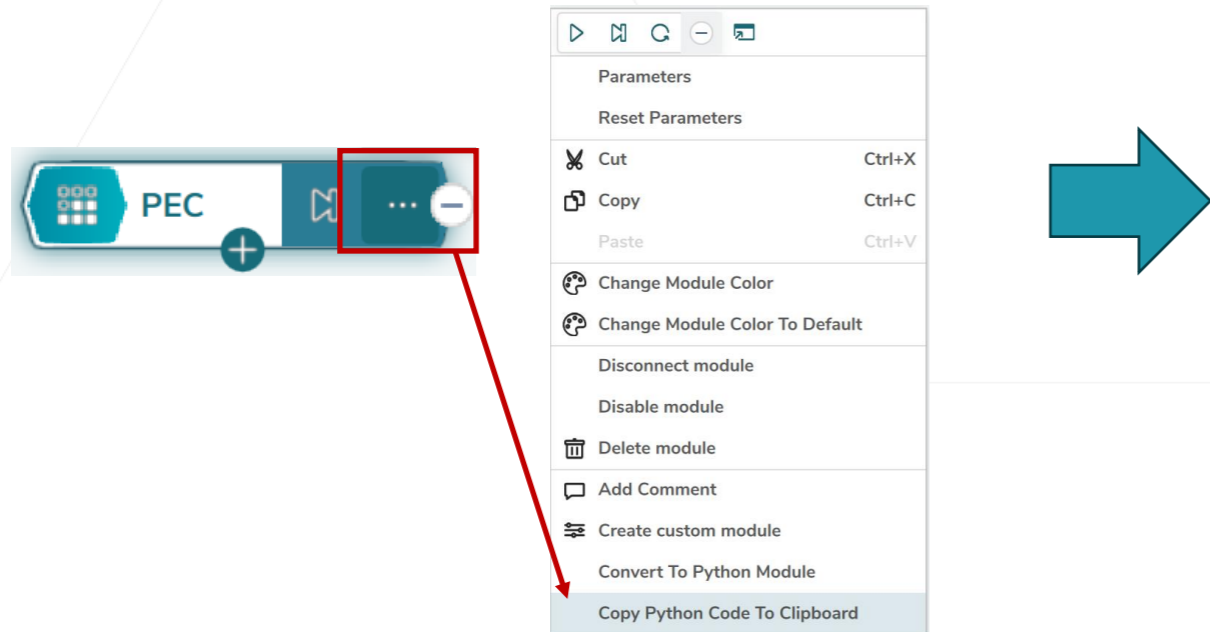
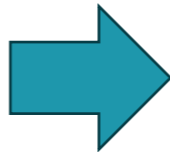
```

1. BEAMER.heal(**gobject**,
2. {'TargetLayer' : '1(0)',
3. 'SoftFrame' : 0.300000,
4. 'HierarchicalProcessing' : True,
5. 'LayerAssignment' : 'AllLayer',
  
```

P-Xor

**in1** and **out1** are **default** names to input and output data from **BEAMER** to **Python**

- Special modules as **Export**, **PEC**, etc. are not listed in the Modules tab, but they have their respective commands! Just copy paste the code into **Python**:





```

1 healing = BEAMER.heal( in1,
2     {'TargetLayer' : '1(0)',
3     'SoftFrame' : 0.300000,
4     'HierarchicalProcessing' : True,
5     'LayerAssignment' : 'AllLayer',
6     'ProcessingMode' : 'Healing'} )
7
8 pec( **gobject**,
9     {'UserdefinedDoseClassFile' : '**filename**',
10    'MinFractureSizeMode' : 'Automatic',
11    'PSFArchiveIdentifierString' : 'Substrate_Si_Thi
12    'PSFType' : 'Gaussian',
13    'PSFFileName' : '',
14    'Alpha' : 0.005000,
15    'Beta' : 30.000000,
16    'Eta' : 0.600000,
17    'Med1Correction' : False,
18    'Med2Correction' : False,
19    'Gamma1' : 0.300000.

```

**IMPORTANT:** With this method, only the command is copied. User needs to add **BEAMER** library!



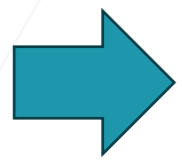
```

8 PEC = BEAMER.pec( healing,
9     {'UserdefinedDoseClassFile' : '**filename**',
10    'MinFractureSizeMode' : 'Automatic',

```

# LAYOUTPy

**CAD software** are the usual alternative for layout creation, but drawing a layout can be **tedious** and **prone to mistakes** if

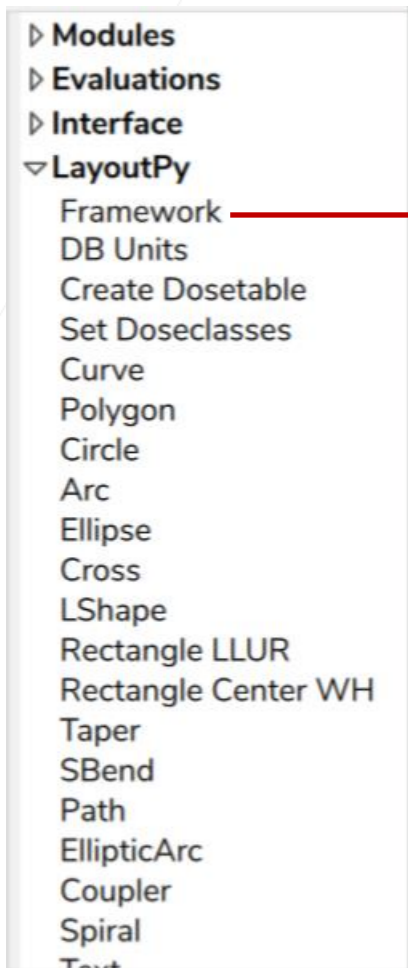


1. large number of shapes are required or
2. elements are irregularly distributed

For instance, **manufacturing augmented and virtual reality devices** with high precision **is challenging** due to the used gratings and our E-beam world working on cartesian coordinates

**BEAMER** and **Python** overcome these problems

- **LAYOUTPy** is a library in **BEAMER** that includes elemental shapes to create simple and elaborated layouts



```
1 from LAYOUTPy import *  
2  
3 db = Database()  
4 with Transaction(db) as txn:  
5     txn.insert( ##Your Object Here## )  
6  
7 db.close()  
8 out1 = db.togobject()
```

The **Framework** gives the general structure to use the **LAYOUTPy** library

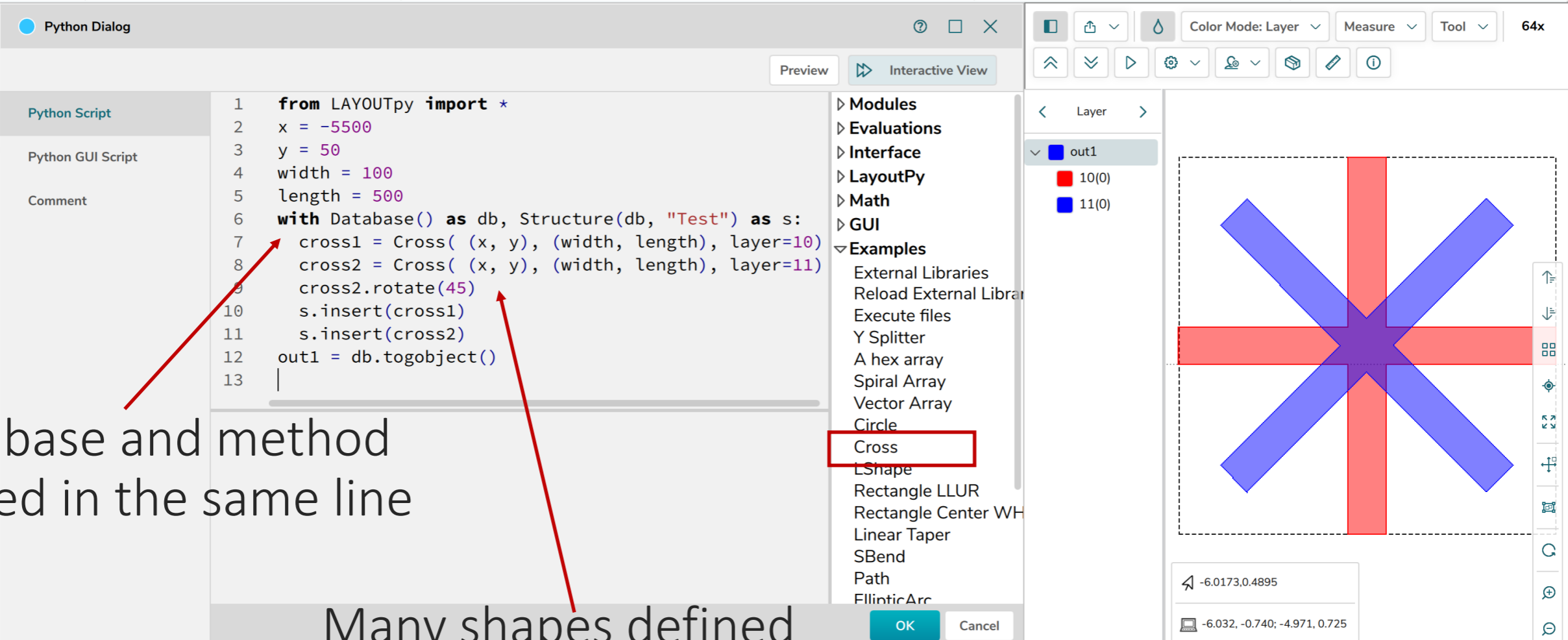
1. Import **LAYOUTPy** library
2. Create a **database** and a **method** for the objects
  - a) Database()
  - b) Transaction() or Structure()
3. Insert the objects
4. Close the database and send output to **BEAMER**

- Drag and drop the object in the **LAYOUTPy** tab to get **general instructions** of the shape (from top to bottom):
  - Structure name and required parameters
  - Description of the structure and parameters
  - Sample command with predefined parameters
  - Additional attributes of the objects and examples

```
1 # txn.insert(Curve([(x1, y1, r1), (x2, y2, r2), (x3, y3, r3), (xn, yn, rn)]))
2 # Inserts a curvature defined by the radius of the curve and the center point of that curve.
3 # At least two sets of points, given by(r, x, y), need to be defined.
4 # Sample
5     txn.insert(Curve([(-5000, 0, 1000), (-5000, 1000, 1000), (-7500, 750, 2000)]))
6     txn.insert(Curve([(-7500, 4000, 1000), (-8500, 3000, 1000), (-9500, 4000, 1000), (-8500, 5000, 1000)]))
7     txn.insert(Curve([(-8500, -2000, 500), (-7500, -2000, 500)]))
8 #---
9 # Additional attributes are : area, bias, invert, mirror, rotate, scale, and translate
10    object = Curve([(-5000, 0, 1000), (-5000, 1000, 1000), (-7500, 750, 2000)])
11    object.rotate(97)
12    object.translate(-5000,0)
13    txn.insert(object)
14 #-- -
15
```

- ▷ Modules
- ▷ Evaluations
- ▷ Interface
- ▼ LayoutPy
  - Framework
  - DB Units
  - Create DoseTable
  - Set DoseClasses
  - Curve**
  - Polygon
  - Circle
  - Arc
  - Ellipse
  - Cross
  - LShape
  - Rectangle LLUR

- In the **Examples** tab the shapes are ready to run (they include the Framework)



The screenshot shows the Python Dialog interface with the following components:

- Python Script:**

```

1  from LAYOUTpy import *
2  x = -5500
3  y = 50
4  width = 100
5  length = 500
6  with Database() as db, Structure(db, "Test") as s:
7      cross1 = Cross( (x, y), (width, length), layer=10)
8      cross2 = Cross( (x, y), (width, length), layer=11)
9      cross2.rotate(45)
10     s.insert(cross1)
11     s.insert(cross2)
12     out1 = db.togobject()
13

```
- Class List:** A list of classes is shown on the right, with 'Cross' highlighted in a red box.
- 3D Visualization:** A 3D view shows a red cross and a blue cross rotated 45 degrees, intersecting at the center. The red cross is labeled '10(0)' and the blue cross is labeled '11(0)' in the layer list.

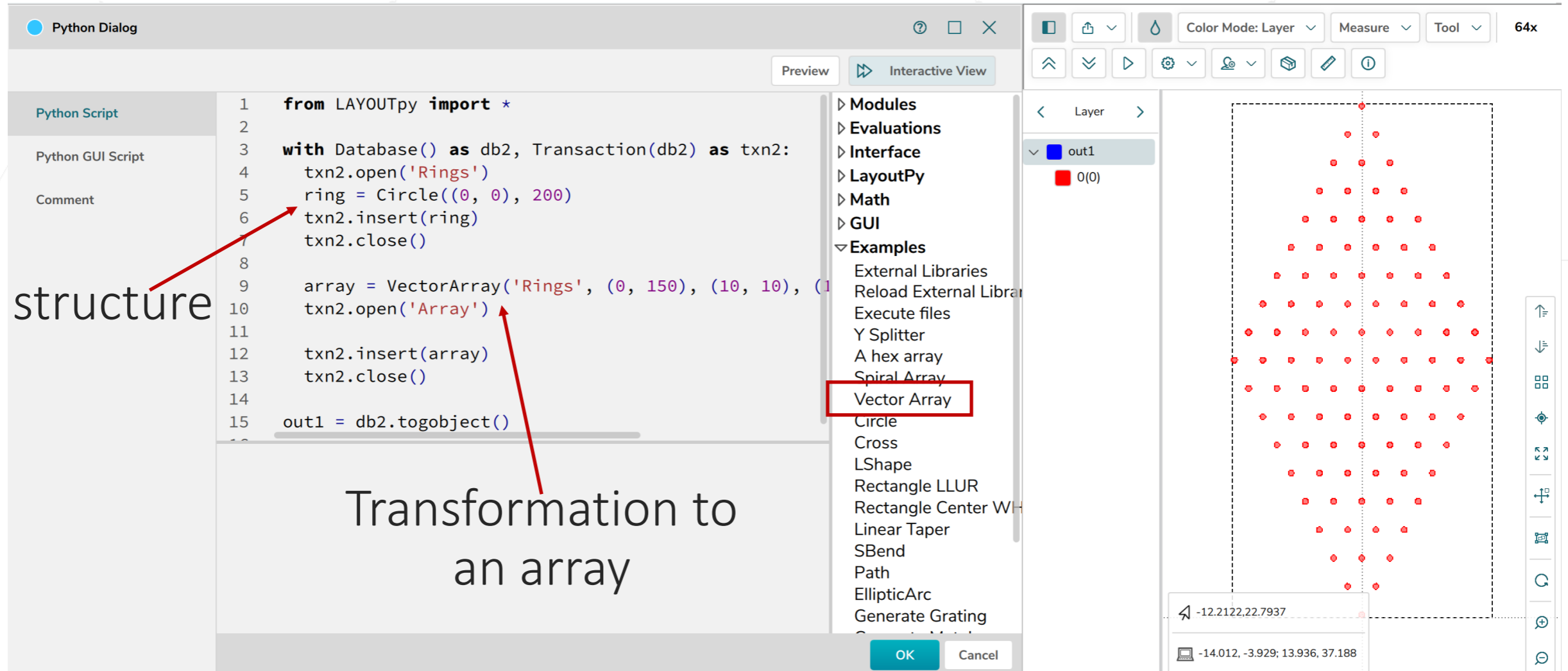
Database and method defined in the same line

Many shapes defined as object

- **LAYOUTPy** includes **arrays** commands to build clustered structures

Base structure

Transformation to an array



```

1  from LAYOUTpy import *
2
3  with Database() as db2, Transaction(db2) as txn2:
4      txn2.open('Rings')
5      ring = Circle((0, 0), 200)
6      txn2.insert(ring)
7      txn2.close()
8
9      array = VectorArray('Rings', (0, 150), (10, 10), (1
10     txn2.open('Array')
11
12     txn2.insert(array)
13     txn2.close()
14
15     out1 = db2.togobject()

```

Python Dialog

Python Script

Python GUI Script

Comment

Preview Interactive View

Color Mode: Layer Measure Tool 64x

Layer

out1

0(0)

Vector Array

Circle

Cross

LShape

Rectangle LLUR

Rectangle Center WH

Linear Taper

SBend

Path

EllipticArc

Generate Grating

OK Cancel

-12.2122,22.7937

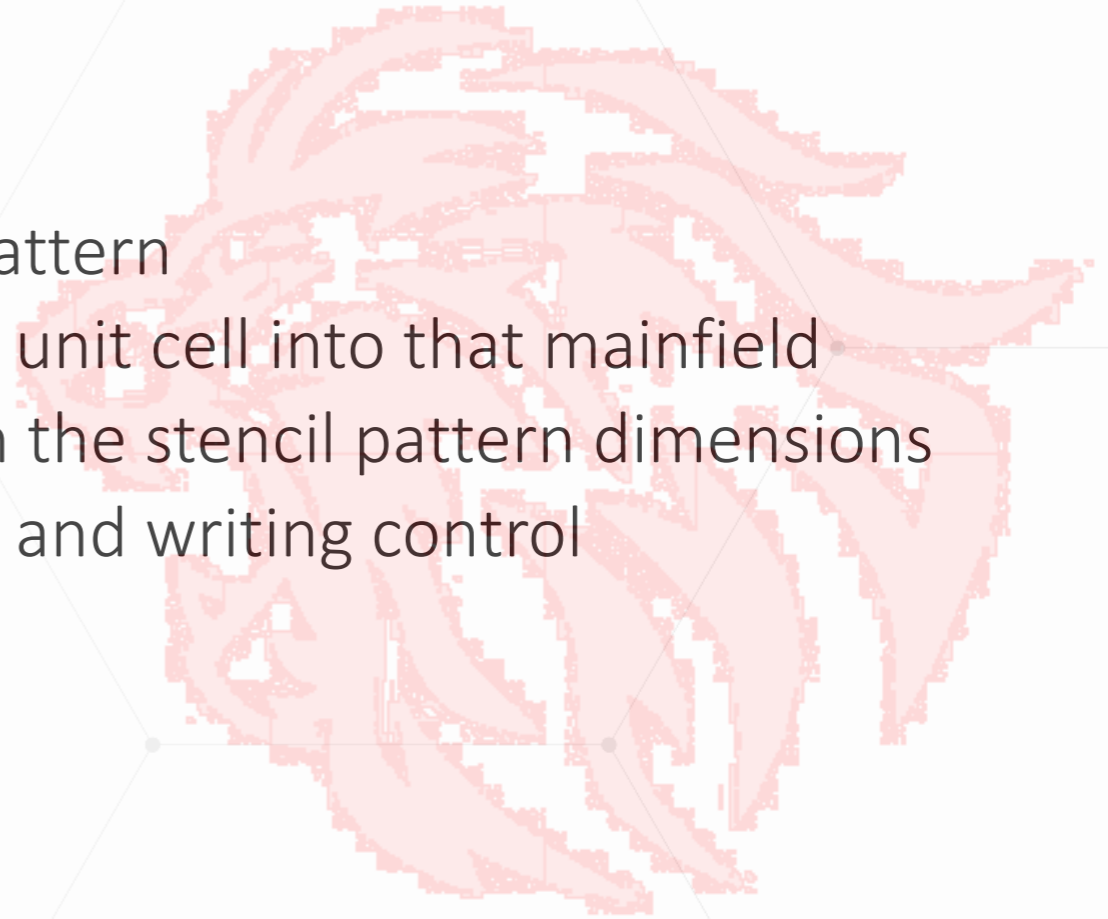
-14.012, -3.929; 13.936, 37.188

# Populating Arbitrary Shapes with Patterns


Photonics is a growing field of applications and often you are faced with the challenge to write large areas with a predefined pattern:

This example shows:

- A way to generate a unit cell like reference pattern
- Take a given main field size and populate the unit cell into that mainfield
- Generate a pattern automatically pending on the stencil pattern dimensions
- Generate the final pattern using field control and writing control

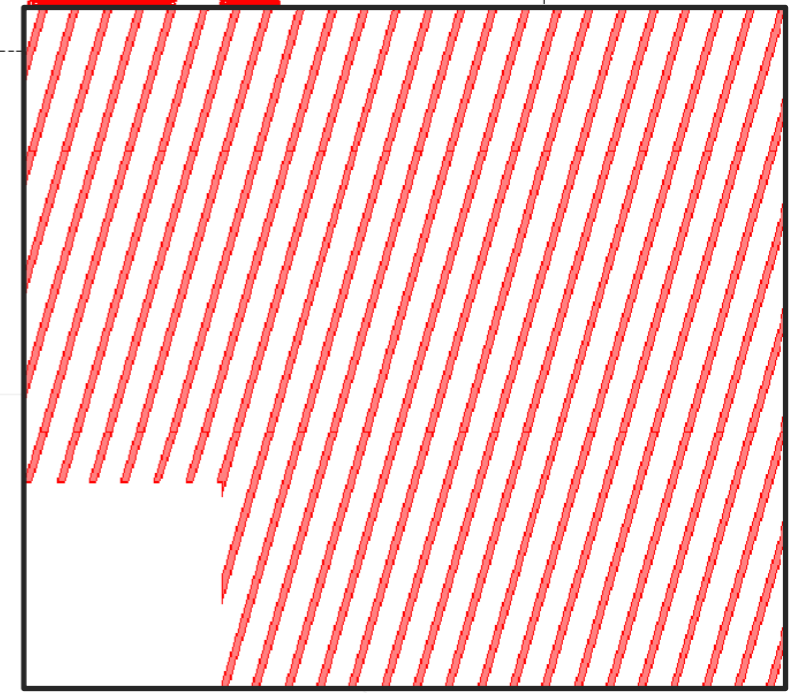
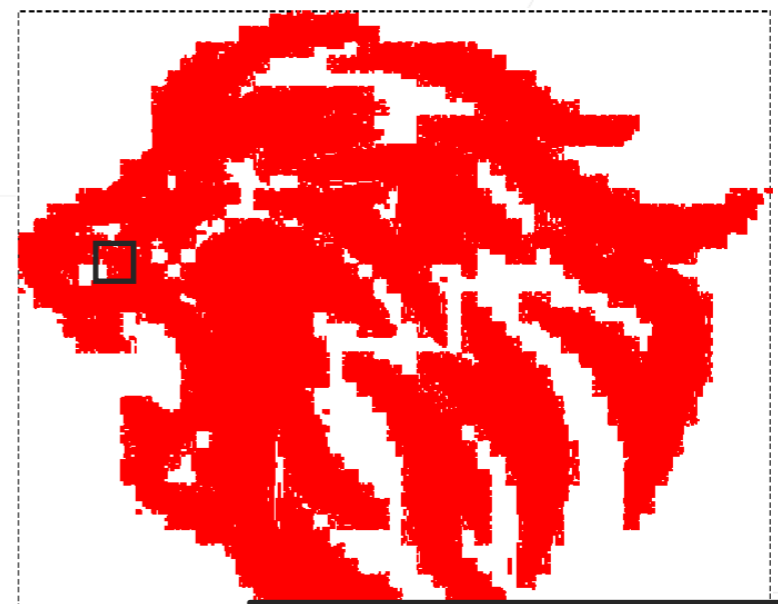


```
1 #
2 # A script to generate from an incoming pattern a grating populated outline
3 #
4 from LAYOUTpy import *
5 import math
6 import numpy as np
7
8
9 # GRATING PARAMETERS #
10
11 # parameters = [[layer, angle [°], width [um], pitch [um]]]
12 parameters = [[10, 25, 0.5, 1], [12, 34, 0.5, 1]]
13
14 #
15 # TOOL SETTINGS
16 #
17 subfieldsize = 4 # Subfield Size in [um]
18 subfield_usage = 0.9 # factor of subfield usage
19 mainfieldsize = 800 # Mainfield Size in [um]
20 res = 0.01 # Resolution in [um]
21 degree45_factor = 0.6 # use this factor in case your angle is 45 degrees to reduce y mainfield size
22 # and by this avoid 0 subfields in x
23 #
24 # Shape mode to be either POLYGONS (0) or PATHS (1) for the grating
25 # Define overlap mode to be true (1) or false(0)
26 #
```

Preview  Interactive View

- ▶ Modules
- ▶ Evaluations
- ▶ Interface
- ▶ LayoutPy
- ▶ Math
- ▶ GUI
- ▼ Examples
  - External Libraries
  - Reload External Libraries
  - Execute files
  - Y Splitter
  - A hex array
  - Spiral Array
  - Vector Array
  - Circle
  - Cross
  - LShape
  - Rectangle LLUR
  - Rectangle Center WH
  - Linear Taper
  - SBend
  - Path
  - EllipticArc
  - Generate Grating**
  - Generate Metalens
  - Runtime Error
- ▶ Snippets
- ▶ History

OK Cancel

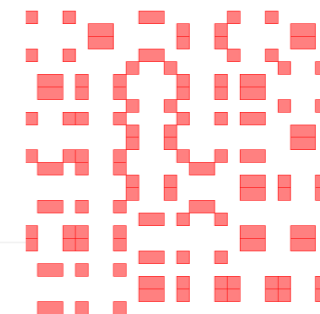
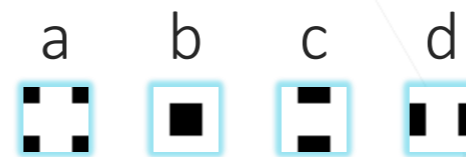


# Application Examples

**Statement:** A set of bitmap patterns needs to be combined into a complex pattern. Each bitmap is a single cell, and the cells are arranged by a user specified scheme. This scheme is stored in a csv file.

**Solution:** With Python we can read the csv file with the arrangement of the bitmaps and compose the arrangement as desired.

	A	B	C	D	E	
1	a	b	c	d	a	b
2	b	d	a	d	b	a
3	a	a	d	a	c	b
4	c	a	d	c	b	d
5	d	d	c	a	b	b
6	c	a	b	d	d	d



- System libraries and **BEAMER** library path that includes the **BEAMERpy** and **csv** libraries
- **BEAMERpy** and **csv** libraries loaded
- Reducing names for simplicity

```
import sys
import os

sys.path.append("C:/Program Files/BEAMER/v7.1.0_x64")

import BEAMERpy
import csv

BEAMER = BEAMERpy.GBEAMER()
```

- Reading the csv file using the function `.reader` and saving it to the `incsv` variable
- Function `.import_bmp` to load the bitmaps
  - `sizeofapixel` sets the size of a converted pixel according to the bitmap
- `,empty'` generates a dummy layout that is needed to merging the bitmaps into the final array

```
incsv = csv.reader(open('t:/work/desktop/csv_layout/load.csv', 'r'), delimiter=';')
```

```
sizeofapixel = 1.000
```

```
bmpa = BEAMER.import_bmp( {'PixelFormat' : sizeofapixel, 'FileName': 't:/work/desktop/csv_layout/a.bmp'})
```

```
bmpb = BEAMER.import_bmp( {'PixelFormat' : sizeofapixel, 'FileName': 't:/work/desktop/csv_layout/b.bmp'})
```

```
bmpc = BEAMER.import_bmp( {'PixelFormat' : sizeofapixel, 'FileName': 't:/work/desktop/csv_layout/c.bmp'})
```

```
bmpd = BEAMER.import_bmp( {'PixelFormat' : sizeofapixel, 'FileName': 't:/work/desktop/csv_layout/d.bmp'})
```

```
empty = BEAMER.extract_layer( bmpa,  
    {'ExtentMode' : 'Default',  
    'LayerSet' : 'anyone'} )
```

- **,bmpsize'** defines the size of the bitmaps in pixels. This helps to calculate the shift between two bitmaps
- **,maxindex\_x/y'** sets the size of the array in the csv file (how many bitmaps in x and y are in the matrix)

```
bmpsize = 4  
maxindex_x = 6  
maxindex_y = 6
```

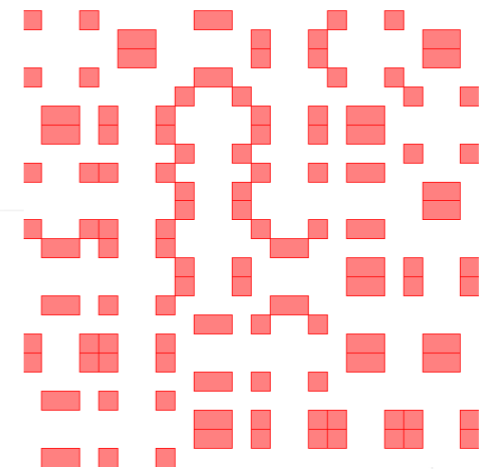
The `,for'` loop processes the csv row by row and merges the bitmaps into one layout. The `,-1'` in the `,shifty'` variable places the row underneath another, `,+1'` would stack them

```
for row in incsv:
    counter_y = 0
    counter_x = 0
    while counter_x < maxindex_x:
        shiftx = bmpsize * sizeofapixel * counter_x
        shifty = -1 * bmpsize * sizeofapixel * counter_y
        print(row[counter_x])
        if row[counter_x] == 'a':
            NE = BEAMER.transform( bmpa, {'CoordinateOrigin' : 'Center', 'ShiftX' : shiftx, 'ShiftY' : shifty} )
        if row[counter_x] == 'b':
            NE = BEAMER.transform( bmpb, {'CoordinateOrigin' : 'Center', 'ShiftX' : shiftx, 'ShiftY' : shifty} )
        if row[counter_x] == 'c':
            NE = BEAMER.transform( bmpc, {'CoordinateOrigin' : 'Center', 'ShiftX' : shiftx, 'ShiftY' : shifty} )
        if row[counter_x] == 'd':
            NE = BEAMER.transform( bmpd, {'CoordinateOrigin' : 'Center', 'ShiftX' : shiftx, 'ShiftY' : shifty} )
        counter_x += 1
    empty = BEAMER.merge( [empty, NE] )
```

The last part positions the layout such that it is in the 1st quadrant of the coordinate system and exports them to the Layout **BEAMER** internal ldb format.

This ldb can be imported to **BEAMER** GUI and exported to a specific tool format from there.

```
empty = BEAMER.transform(empty, {'CoordinateOrigin': 'BottomLeft'})  
gobj_1 = BEAMER.export_ldb(empty, { 'FileName' : os.path.join(os.getcwd(), 'bitmap_export.ldb') } )
```



# LAYOUTPy in Python IDE Example

- **LAYOUTPy** library exists in the same path as **BEAMERpy** →
- The examples can be run similarly as in the **BEAMER GUI**, but the library needs to **explicitly** be part of the command →
- Other **BEAMER** modules can be added to modify and export → the layout

```

import os
import sys

sys.path.append("C:/Program Files/BEAMER/v7.1.0_x64")

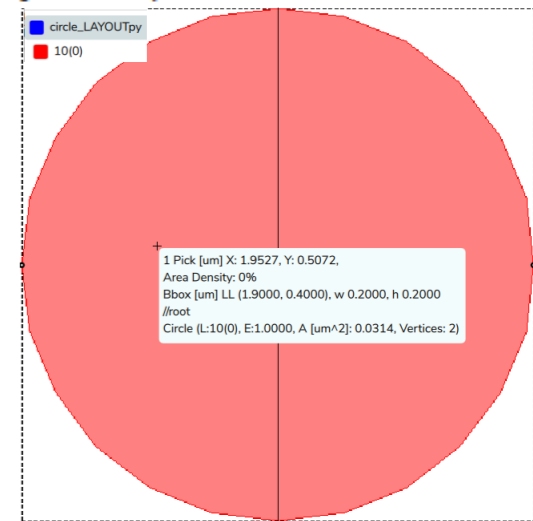
import BEAMERpy
import LAYOUTpy

BEAMER = BEAMERpy.GBEAMER()

x = 2000
y = 500
radius = 100
circle1 = LAYOUTpy.Circle( (x, y), radius, layer=10)

db = LAYOUTpy.Database()
with LAYOUTpy.Transaction(db) as txn:
    txn.insert(circle1)
db.close()
out1 = db.togobject()

gobj_0 = BEAMER.heal( out1,
    {'TargetLayer' : '1(0)',
     'ProcessingMode' : 'Healing'} )
gobj_1 = BEAMER.export_gds(gobj_0,
    { 'FileName' : os.path.join(os.getcwd(), 'circle.gds') } )
    
```



- BEAMER and **Python** working together **strengthens** the device **manufacturing processes** from layout design to exposure writing order
- The **Python** scripting interface **simplifies** the **automation** of layout design and preparation by fading out the boundaries between exposure-tool and **BEAMER**
- The library **LAYOUTPy** extends **BEAMER-functionality** by providing native exposure-tool structures
- **LAYOUTPy** provides alternatives to create **sophisticated layouts**

# Thank You!

[help@genisys-gmbh.com](mailto:help@genisys-gmbh.com)



## Headquarters

GenISys GmbH  
Inselkammerstr. 5  
D-82008 Unterhaching (Munich)  
GERMANY

☎ +49 (0)89 954 5364 0

☎ +49 (0)89 954 5364 99

✉ [info@genisys-gmbh.com](mailto:info@genisys-gmbh.com)

## USA Office

GenISys Inc.  
P.O. Box 410956  
San Francisco, CA  
94141-0956  
USA

☎ +1 (408) 353 3951

✉ [usa@genisys-gmbh.com](mailto:usa@genisys-gmbh.com)

## Japan / Asia Pacific Office

GenISys K.K.  
German Industry Park  
1-18-2 Hakusan Midori-ku  
Yokohama 226-0006  
JAPAN

☎ +81 (45) 530 3306

☎ +81 (45) 532 6933

✉ [apsales@genisys-gmbh.com](mailto:apsales@genisys-gmbh.com)