GDSII Scripting for Lithography

Why script your layout?

For most lithography work at CNF you should provide your designs in the GDSII format (`.gds`). Direct-write tools, such as the JEOL EBL systems and the Heidelberg mask writers, use these layouts in their job files.

To create your layouts, you can use CAD software such as KLayout and LEdit, both of which are available on CNF workstations. KLayout is free and cross-platform, so you can prepare your files ahead of time, before you come to CNF for fabrication. CAD software can handle complex shapes and large arrays, and may be all you need for your purposes.

On occasion, however, you may find that it is tedious to draw your layout in an editor, for example because you need a large number of different shapes, or because elements are irregularly distributed, or simply because the large number of steps involved makes mistakes likely. Scripting is a good alternative in this cases. This page will show you how to do it in Python via the PHIDL package.

Preparations

You should know basic Python, including environment setup, command line use and package installation, although some of this is repeated below. If you do not know any Python, it may still be worthwhile to learn if you plan to run several projects at CNF, and it is of course a useful language in many other applications.

Important note

Pay close attention to specific machine requirements for GDSII files. A notable limitation is on polygon vertex number (the maximum supported by the Heidelbergs is 199). If you require better detail than this can allow (most likely with curved shapes) you will need to break up your polygons in smaller chunks.

Installing Python

Python is already installed on Korat and Minx.

You can download the latest Python release from the official website, or use a package manager (for example, your distribution's default in Linux or Homebrew in macOS). Make sure `python` is accessible by terminal. Using an IDE, such as Visual Studio Code (free), is recommended.

If you manage multiple projects and are hesitant to modify your environment, you can for example use pyenv.

Installing PHIDL

Phidl is already installed on Korat and Minx. See the "Running on Korat and Minx" section below for more information.

Phidl is available with Python's `pip` package manager. In the command line use (you will first have to have installed the python devel packages from your linux distribution):

```
pip3 install phidl
```

In addition to PHIDL, you may need maths packages to calculate your device's geometry. The most common such package is `numpy`.

Note the above command requires root privileges as it will attempt to install phidl and requirements system-wide. Instead, we recommend you install into your local home directory to avoid creating issues with the system python3 packages. use the following command:

```
pip3 install --user phidl
```

You will then need to add the install directory to your PYTHONPATH. Most likely, you will need to add the directory: ~/.local/lib/python3.6/site-packages

Scripting a GDSII file

Although this page should be sufficient to get you started, it's well worth it to have a look at the full PHIDL documentation. The tutorials are a great place to learn its various functions. The geometry reference lists all inbuilt geometry types, including important aids such as text and and lithography test structures. Make sure you go through its contents: the inbuilt types can save you a large amount of time.

The examples below are only one possible way of doing things. Refer to the documentation and find the way that works best for you.

Saving and running scripts

Save your scripts with the `.py` extension. You can run them in the command line with

```
python script.py
```

or, if you have `ipython` installed,

```
ipython
run script
```

Basic grating

We'll start by writing a script that creates a GDSII file containing a basic grating. First, the packages must be imported. For now, we only need `Device` to create the layout and `quickplot` to view it.

```
from phidl import Device
from phidl import quickplot as qp
```

Then, we define our grating's dimensions. *Note that, in all examples, dimensions are very much unrealistic and chosen for ease of viewing.*

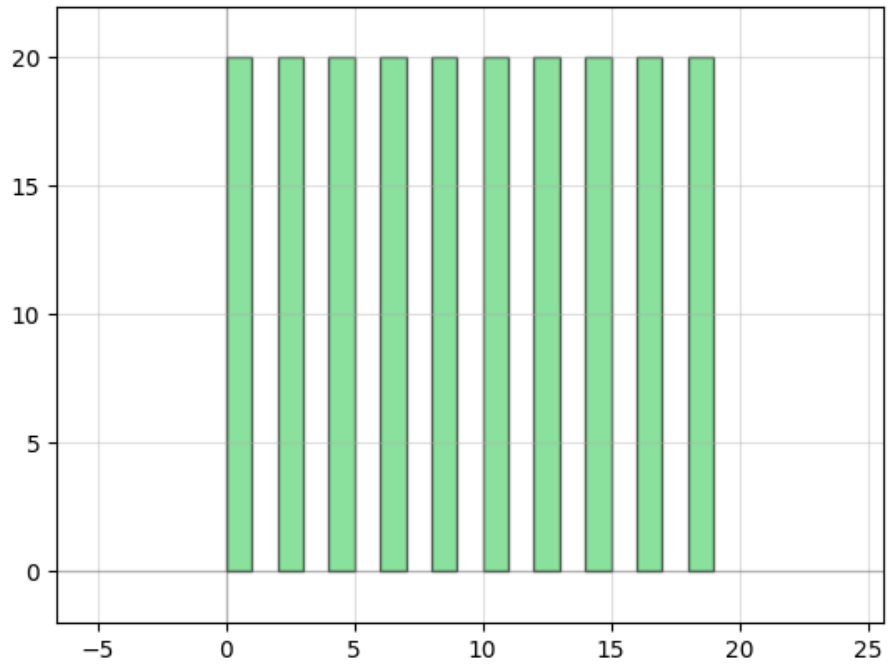
```
N_X = 10    # Number of lines
P_X = 2     # Grating period, um
W_X = 1     # Line width, um
W_Y = 20    # Line length, um
```

Now we can create an empty device and use a loop to add polygons to it according to our specifications.

```
D = Device()

for x in range(0, N_X):
    p0 = (x * P_X, 0)
    p1 = (x * P_X + W_X, 0)
    p2 = (x * P_X + W_X, W_Y)
    p3 = (x * P_X, W_Y)
    D.add_polygon([p0, p1, p2, p3])
```

That's it. We can plot our layout right away, which is a very useful troubleshooting step.

```
qp(D) # Plot device
```



Finally, we can save the layout to a `.gds` file.

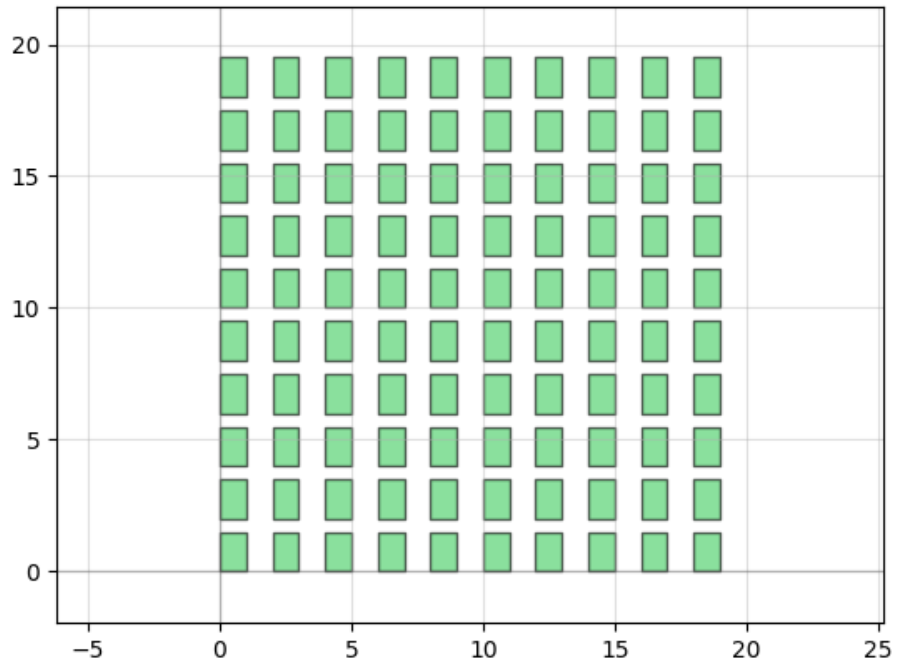
```
D.write_gds('grating')
```

Note that repeated structures can be also made by using GDSII references, as outline in the references section below.

Basic array

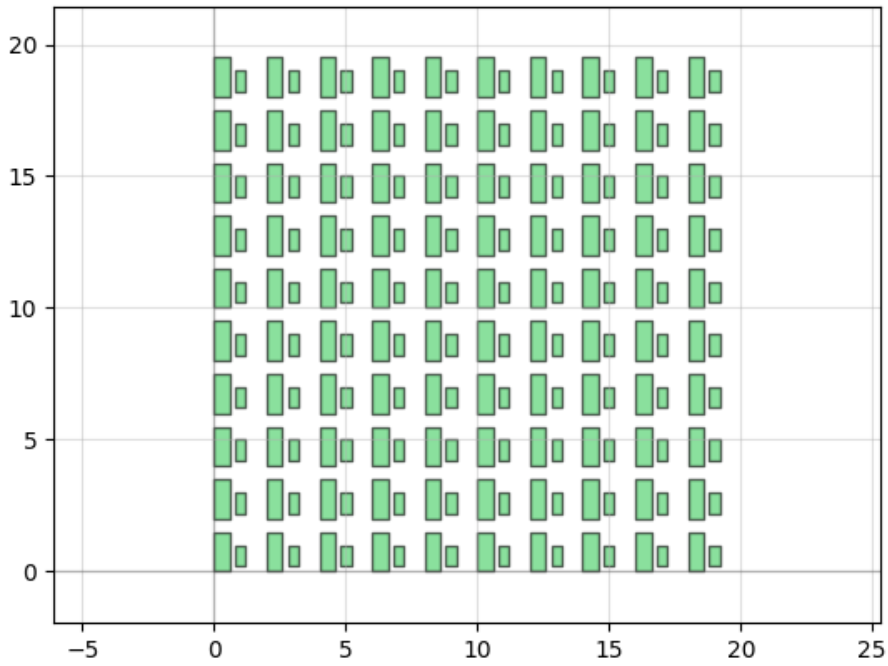
It is easy to modify the above example to have a two-dimensional array of identical structures.

```
from phidl import Device
from phidl import quickplot as qp

N_X = 10     # Number of structures along x
N_Y = 10     # Number of structures along y
P_X = 2      # Array period along x, um
P_Y = 2      # Array period along y, um
W_X = 1      # Structure dimension along x, um
W_Y = 1.5    # Structure dimension along y, um

D = Device()

for x in range(0, N_X):
    for y in range(0, N_Y):
        p0 = (x * P_X, y * P_Y)
        p1 = (x * P_X + W_X, y * P_Y)
        p2 = (x * P_X + W_X, y * P_Y + W_Y)
        p3 = (x * P_X, y * P_Y + W_Y)
        D.add_polygon([p0, p1, p2, p3])

qp(D)    # Plot device

D.write_gds('array')
```



Because we define each vertex of our polygons, they needn't be rectangular, nor limited to four vertices, so the code above can easily be modified to make arrays out of units with many different shapes. If the unit cell is made up of multiple separate shapes, more polygons can be added during each loop:

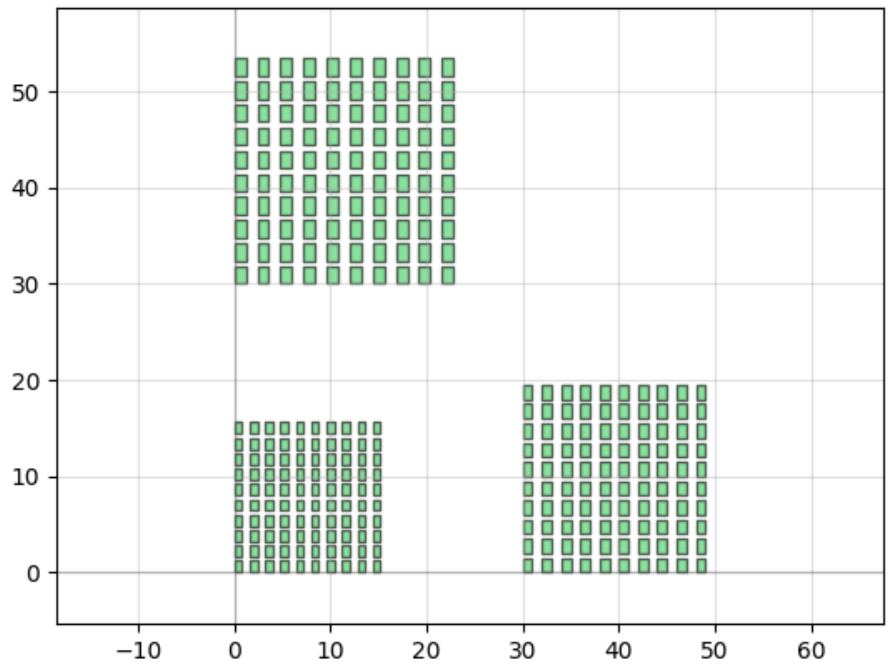
```
from phidl import Device
from phidl import quickplot as qp

N_X = 10     # Number of structures along x
N_Y = 10     # Number of structures along y
P_X = 2      # Array period along x, um
P_Y = 2      # Array period along y, um
W_X = 1      # Structure dimension along x, um
W_Y = 1.5    # Structure dimension along y, um

D = Device()

for x in range(0, N_X):
    for y in range(0, N_Y):
        p0 = (x * P_X, y * P_Y)
        p1 = (x * P_X + W_X, y * P_Y)
        p2 = (x * P_X + W_X, y * P_Y + W_Y)
        p3 = (x * P_X, y * P_Y + W_Y)
        D.add_polygon([p0, p1, p2, p3])

qp(D)    # Plot device

D.write_gds('array')
```



Scaling arrays

It is often convenient to fabricate multiple versions of the same array in (slightly) different sizes, to account for fabrication deviations or mismatches between numerical simulation and experiment. The basic array code from earlier can be modified to create multiple differently scaled arrays. Using a two-dimensional list for the scale factors is a simple way to arrange the arrays in rows, which we do to save space.

```
from phidl import Device
from phidl import quickplot as qp

N_X = 10     # Number of structures along x
N_Y = 10     # Number of structures along y
P_X = 2      # Array period along x, um
P_Y = 2      # Array period along y, um
S_X = 30     # Separation between array origins along x, um
S_Y = 30     # Separation between array origins along y, um
W_X = 1      # Structure dimension along x, um
W_Y = 1.5    # Structure dimension along y, um

SF = [[0.8, 1.], [1.2]]    # Scale factors by row of arrays

D = Device()

for x in range(0, N_X):
    for y in range(0, N_Y):
        for ri, r in enumerate(SF):
            for sfi, sf in enumerate(r):
                p0 = (sf * (x * P_X) + sfi * S_X,
                      sf * (y * P_Y) + ri * S_Y)
                p1 = (sf * (x * P_X + W_X) + sfi * S_X,
                      sf * (y * P_Y) + ri * S_Y)
                p2 = (sf * (x * P_X + W_X )+ sfi * S_X,
                      sf * (y * P_Y + W_Y) + ri * S_Y)
                p3 = (sf * (x * P_X) + sfi * S_X,
                      sf * (y * P_Y + W_Y) + ri * S_Y)
                D.add_polygon([p0, p1, p2, p3])

qp(D)    # Plot device

D.write_gds('arrays')
```



When you fabricate a real device, in addition to having much larger arrays, you will probably use many more scale factors.

Labelling

Labelling is very important both for yourself, to aid in various fabrication and metrology steps, and for experimenters who characterize your device, to help them find structures and make sure they are looking at the right one. In the case of the example above, we should at least label the various arrays so that they are identified unambiguously, and add a sample label as well.

The labels we add here would be very small and difficult to read under low magnification. It is better if essential identifiers, such as row and column letters and numbers, are a hundred or more microns tall: this will make them visible to the naked eye and readable with any microscope. When in doubt, go with few letters (so that overall writing time is not increased excessively) and make them large.

Adding labels is easily done with `text` from PHIDL's geometry library. The default font is suitable for lithography, but of course it will still be unreadable if the letters are too small for the fabrication process you're using.

After creating text, we will need to `move` it to the right place, before adding it to the device with `add_ref`. More on references later. These command can be used with other geometry, too.

Starting from the same code as the previous section, we need to modify the imports to include the PHIDL geometry library:

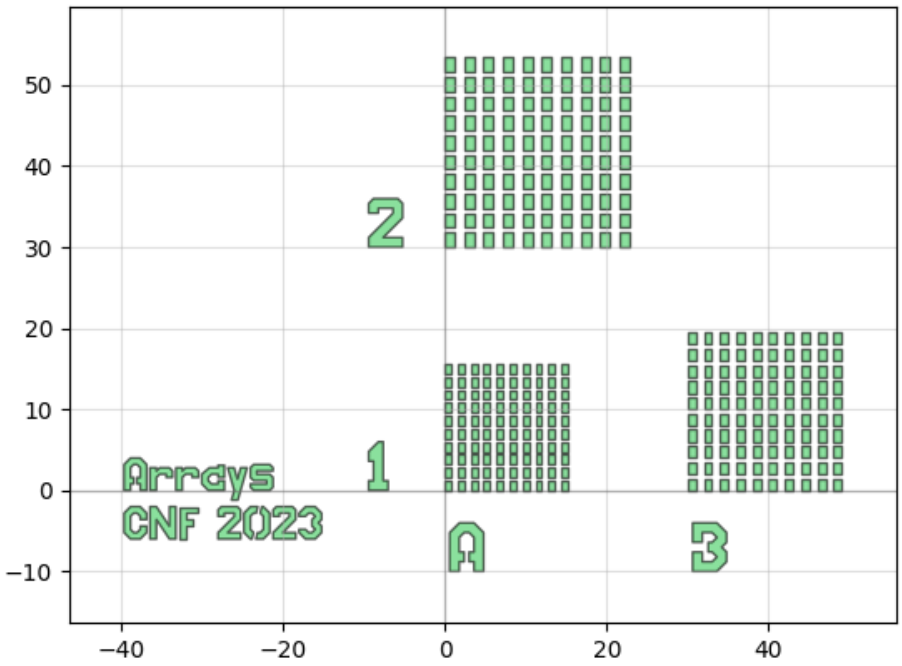
```
from phidl import Device
from phidl import quickplot as qp
import phidl.geometry as pg
```

Then, we add the following code between the `for` loop and the `qp` command to add a label for our device:

```
### Add device identifier

deviceTextStr = 'Arrays\nCNF 2023'
deviceText = pg.text(deviceTextStr, size = 4)
deviceText.move([-40, 0])
D.add_ref(deviceText)
```

Finally, we add code for column and row identifiers just below the previous snippet:

```
### Add row and column labels

TEXT_SIZE = 6

colLabelOffset = -10
rowLabelOffset = -10
colStr = ['A', 'B']
rowStr = ['1', '2']

for ics, cs in enumerate(colStr):
    labelText = pg.text(cs, size = TEXT_SIZE)
    labelText.move([ics * S_X, colLabelOffset])
    D.add_ref(labelText)

for irs, rs in enumerate(rowStr):
    labelText = pg.text(rs, size = TEXT_SIZE)
    labelText.move([rowLabelOffset, irs * S_Y])
    D.add_ref(labelText)
```



Once again, in an actual device both arrays and labels would be much larger.

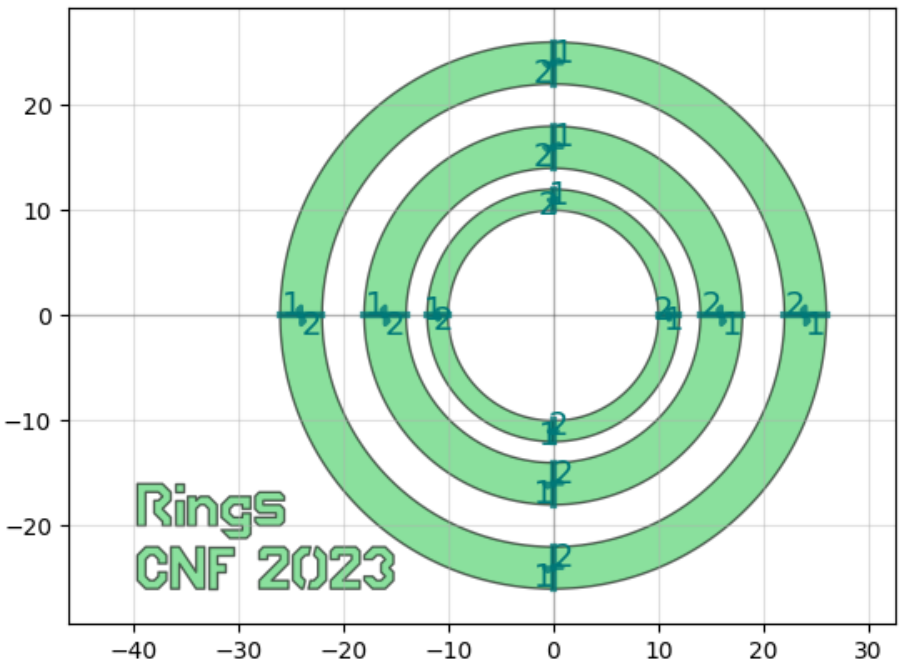
Note that we have added the labels without modifying the pattern's origin. If you want the origin somewhere else you must change the various position shifts to account for it, or use `move` as shown in the origin section below.

Circles and arcs

The GDSII format does not support curves, which means that curved objects are polygons, too, albeit likely with large vertex counts. When working with lots of vertices, remember not to exceed the target tool's limit.

Let's say our device made up of three rings, each defined by an inner and outer radius. Moreover, let's ask that our rings be approximated by polygons with one vertex per degree, although this is far too much for the rings in this example. This means 2 × 360 vertices, which for example is much larger than the 199 limit for the Heidelbergs. We can't use PHIDL's `ring` and must instead use four instances of `arc`. One possible way of doing so is below.

```
from phidl import Device
from phidl import quickplot as qp
import phidl.geometry as pg

ARC_START = [0, 90, 180, 270]
ARC_EXTENT = 90
DEG_PER_VERT = 1

R = [[10, 12],[14, 18],[22, 26]] # Inner and outer radius, um

D = Device()

for r in R:
    r0 = r[0]
    r1 = r[1]
    for a in ARC_START:
        arc = pg.arc(radius = (r1+r0)/2,
                width = (r1-r0),
                theta = ARC_EXTENT,
                start_angle = a,
                angle_resolution = DEG_PER_VERT)
        D.add_ref(arc)

### Add device identifier

deviceTextStr = 'Rings\nCNF 2023'
deviceText = pg.text(deviceTextStr, size = 4)
deviceText.move([-40, -20])
D.add_ref(deviceText)

qp(D)    # Plot device

D.write_gds('rings')
```



The objects marked "1" and "2" are ports, which are used by PHIDL's routing and are a possible alternative to the method in this example. Because each ring is four arcs, although it should not be a problem here, a *healing* step is recommended in the job preparation software (it always is).

Layers

Layers are a useful GDSII feature that is especially important in multistep processes. However, consider whether you do need multiple layers in the final GDSII file, or you should be laying out the various steps side by side on the same layer instead (which is often more straightforward when making a photomask).

PHIDL supports GDSII layers with extensive utilities to handle them. GDSII layers are defined by two integers between 0 and 255, the *layer number* and *dat atype*. If the datatype is not specified, PHIDL sets it to 0.

Suppose we want to add another layer to our arrays device above. It could represent a second lithography step for the purpose of etching a multi-step profile on the device. Keep in mind that, in an actual device, this would require alignment marks appropriate to the tool being used.

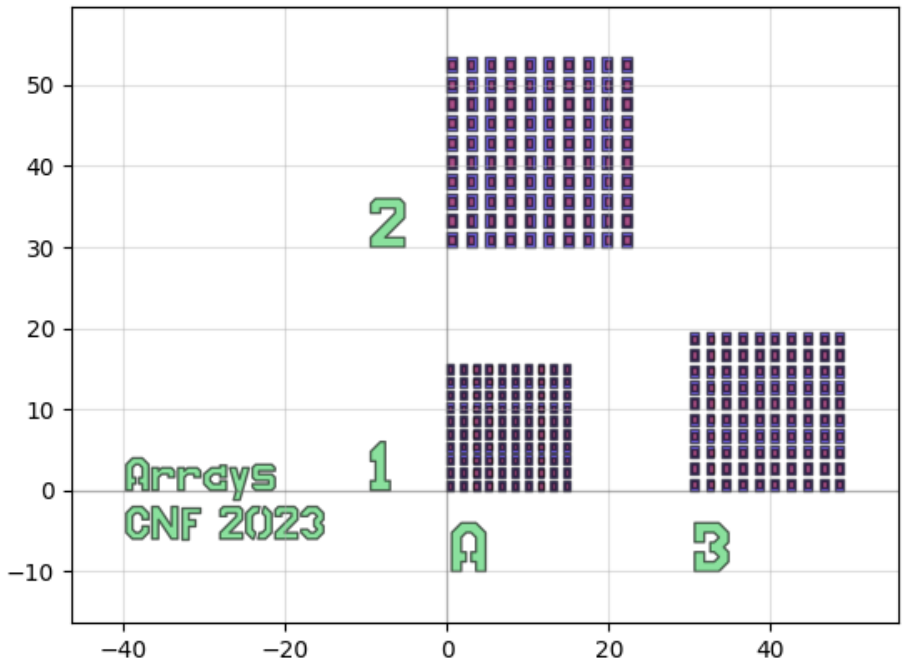
We must modify our import to

```
from phidl import Device, Layer
```

and define three layers with `layer0 = (0,0)`, etc. We then assign (somewhat arbitrarily) the text to layer 0, our previous structures to layer 1 and the second-step structures to layer 2 by specifying the `layer` keyword when adding geometry.

```
from phidl import Device, Layer
from phidl import quickplot as qp
import phidl.geometry as pg

N_X = 10      # Number of structures along x
N_Y = 10      # Number of structures along y
P_X = 2       # Array period along x, um
P_Y = 2       # Array period along y, um
S_X = 30      # Separation between array origins along x, um
S_Y = 30      # Separation between array origins along y, um
W1_X = 1      # Structure dimension along x, um
W1_Y = 1.5    # Structure dimension along y, um
W2_X = 0.6    # Second layer structure dimension along x, um
W2_Y = 0.9    # Second layer structure dimension along y, um

SF = [[0.8, 1.], [1.2]]    # Scale factors by row of arrays

D = Device()

layer0 = (0, 0)
layer1 = (1, 0)
layer2 = (2, 0)

wDiffX = (W1_X - W2_X) / 2
wDiffY = (W1_Y - W2_Y) / 2

for x in range(0, N_X):
    for y in range(0, N_Y):
        for ri, r in enumerate(SF):
            for sfi, sf in enumerate(r):
                for u, v, du, dv, l in zip([W1_X, W2_X], [W1_Y, W2_Y],
                                           [0, wDiffX], [0, wDiffY],
                                           [layer1, layer2]):
                    p0 = (sf * (x * P_X + du) + sfi * S_X,
                          sf * (y * P_Y + dv) + ri * S_Y)
                    p1 = (sf * (x * P_X + du + u) + sfi * S_X,
                          sf * (y * P_Y + dv) + ri * S_Y)
                    p2 = (sf * (x * P_X + du + u )+ sfi * S_X,
                          sf * (y * P_Y + dv + v) + ri * S_Y)
                    p3 = (sf * (x * P_X + du) + sfi * S_X,
                          sf * (y * P_Y + dv + v) + ri * S_Y)
                    D.add_polygon([p0, p1, p2, p3], layer = l)

### Add device identifier

deviceTextStr = 'Arrays\nCNF 2023'
deviceText = pg.text(deviceTextStr, size = 4, layer = layer0)
deviceText.move([-40, 0])
D.add_ref(deviceText)
```

```
### Add row and column labels

TEXT_SIZE = 6

colLabelOffset = -10
rowLabelOffset = -10
colStr = ['A', 'B']
rowStr = ['1', '2']

for ics, cs in enumerate(colStr):
    labelText = pg.text(cs, size = TEXT_SIZE)
    labelText.move([ics * S_X, colLabelOffset])
    D.add_ref(labelText)

for irs, rs in enumerate(rowStr):
    labelText = pg.text(rs, size = TEXT_SIZE)
    labelText.move([rowLabelOffset, irs * S_Y])
    D.add_ref(labelText)

qp(D)    # Plot device

D.write_gds('arrays')
```



The three colors identify the three layers.

There's much more that PHIDL can do with layers, which could be useful if your device is particularly complex. When using many layers, keep a consistent nomenclature across devices.

Using references

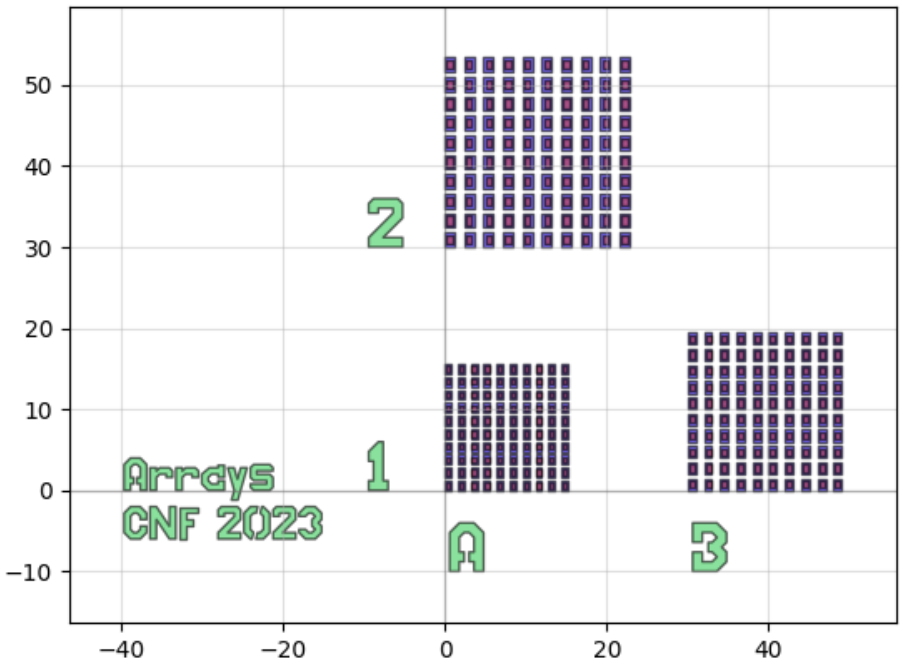
PHIDL can be used to access the inbuilt GDSII reference system. So far, we have been using `add_ref` just for adding text, but it is much more powerful. It is very useful to create multiple identical shapes by reusing the same basic geometry rather than creating new geometry over and over again.

Let's take the example from the previous section and rewrite it using references. Additionally, instead of using a `for` loop above to create arrays, let's make use of the inbuilt `add_array`.

```
from phidl import Device, Layer
from phidl import quickplot as qp
```

```
import phidl.geometry as pg

N_X = 10      # Number of structures along x
N_Y = 10      # Number of structures along y
P_X = 2       # Array period along x, um
P_Y = 2       # Array period along y, um
S_X = 30      # Separation between array origins along x, um
S_Y = 30      # Separation between array origins along y, um
W1_X = 1      # Structure dimension along x, um
W1_Y = 1.5    # Structure dimension along y, um
W2_X = 0.6    # Second layer structure dimension along x, um
W2_Y = 0.9    # Second layer structure dimension along y, um

SF = [[0.8, 1.], [1.2]]    # Scale factors by row of arrays

D = Device()

layer0 = (0, 0)
layer1 = (1, 0)
layer2 = (2, 0)

wDiffX = (W1_X - W2_X) / 2
wDiffY = (W1_Y - W2_Y) / 2

for ri, r in enumerate(SF):
    for sfi, sf in enumerate(r):
        P1 = Device()
        P2 = Device()
        p0 = (0, 0)
        p1 = (sf * W1_X, 0)
        p2 = (sf * W1_X, sf * W1_Y)
        p3 = (0, sf * W1_Y)
        P1.add_polygon([p0, p1, p2, p3], layer = layer1)
        p0 = (0, 0)
        p1 = (sf * W2_X, 0)
        p2 = (sf * W2_X, sf * W2_Y)
        p3 = (0, sf * W2_Y)
        P2.add_polygon([p0, p1, p2, p3], layer = layer2)
        A1 = Device()
        A2 = Device()
        arrayPos = [sfi * S_X, ri * S_Y]
        layerOffset = [sf * wDiffX, sf * wDiffY]
        spacing = [sf * P_X, sf * P_Y]
        A1.add_array(P1, columns = N_X, rows = N_Y,
                     spacing = spacing).move(arrayPos)
        A2.add_array(P2, columns = N_X, rows = N_Y,
                     spacing = spacing).move(
                               layerOffset).move(arrayPos)
        D.add_ref(A1)
        D.add_ref(A2)

### Add device identifier

deviceTextStr = 'Arrays\nCNF 2023'
deviceText = pg.text(deviceTextStr, size = 4, layer = layer0)
deviceText.move([-40, 0])
D.add_ref(deviceText)

### Add row and column labels

TEXT_SIZE = 6

colLabelOffset = -10
rowLabelOffset = -10
colStr = ['A', 'B']
rowStr = ['1', '2']

for ics, cs in enumerate(colStr):
    labelText = pg.text(cs, size = TEXT_SIZE)
    labelText.move([ics * S_X, colLabelOffset])
    D.add_ref(labelText)
```

```
for irs, rs in enumerate(rowStr):
    labelText = pg.text(rs, size = TEXT_SIZE)
    labelText.move([rowLabelOffset, irs * S_Y])
    D.add_ref(labelText)

qp(D)    # Plot device

D.write_gds('arrays')
```

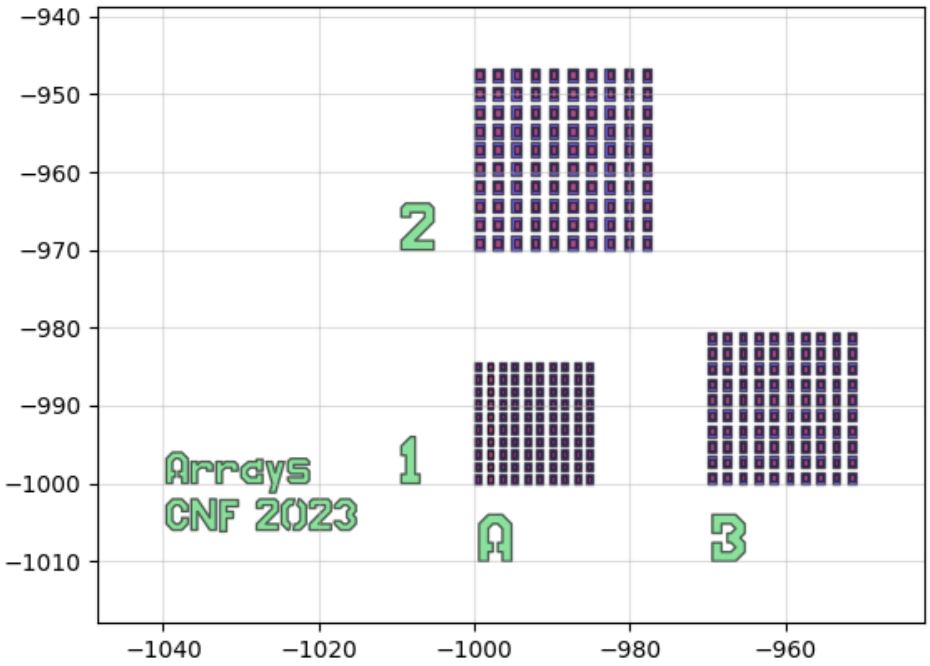


The code is actually longer in this instance, but probably easier to read.

Moving the origin

We have seen the command `move` used to move objects and devices (PHIDL also includes rotating, mirroring and more) and one possible use case for moving your entire, final device is to change the GDSII origin.

```
D.move([-1000, -1000])
```



(Note the axes scales.)

That said, job file preparation software for e-beam and photolithography (such as LayoutBEAMER, or that on the Heidelberg mask writers workstation) wll let you move a pattern's origin wherever you need it.

Running on Korat and Minx

PHIDL and necessary packages have been installed on CNF's Korat and Minx servers, which are accessible from ThinLinc clients in the cleanroom, CAD room or remotely through the Cornell VPN. Links to Korat and Minx terminals are found in the "Applications > CNF" menu. In either terminal, `cd` to the folder containing your script file(s), e.g. `filename.py`, and type:

```
bash
export PYTHONPATH=/usr/local/phidl/lib64/python3.6/site-packages:/usr/local/phidl/lib/python3.6/site-packages:$PYTHONPATH
ipython3 -i filename.py
```

If your code is structured like the examples above, a `.gds` file will be saved, and a plot of your layout will be displayed.

Alternatives to PHIDL

Another CNF resource is *JetStream*, a Java GDSII library. Read the Getting Started guide and the documentation.

Tips and tricks

Follow these practices to make your own life, and that of whoever will characterize your device, much easier:

- Be consistent with labels and nomenclature across devices and GDSII files.
- Label structures and samples clearly, and make labels large enough for experimenters to see by eye and read with low magnification.
- **Include alignment marks** if you require multiple exposures. These vary by tool and are not included in the PHIDL libraries. Download them from each tool page in the CNF users' website and separately add them to your layout files using your GDSII editor of choice.
- Include lithography test structures, especially when running exposure tests.