



Sorbonne Université

lip6 Laboratory

# Hurricane+Python Tutorial

Jean-Paul Chaput
Jean-Paul.Chaput@lip6.fr

**First, a small disclaimer.** This tutorial assumes that you are already familiar with the concepts of vlsi designs, such as *netlist*, *layout*, *instances* and hierarchical design.

## Contents

# 1. Introduction

This tutorial is aimed at two goals :

- Presenting how to use Python scripts to control Coriolis.
- Make a basic introduction about the Hurricane database and its concepts.

While this tutorial is aimed at presenting the Hurricane database, do not feel limited by it. You can use Hurricane objects as attributes of Python objects or use Python containers to store them. The only limitation is that you may not use Hurricane classes as base classes in Python.

All Hurricane objects implements the Python `__str__()` function, they print the result of the C++ method `::getString()`.

## 1.1 Terminology

In the Hurricane database, the *logical* (netlist) and *physical* (layout) views are fused. As the main goal of the database is to support place & route tools, we usually starts with a *pure* netlist which is progessively enriched to become a layout. Cell, in particular, is able to be in any intermediate state. Many of our objects have self-explanatory names, but some don't. Thus we summarize below the more important ones:

| Class | Meaning |
|---|---|
| Cell | The model. A Cell does not have terminals, only nets flagged as *external* |
| Instance | An instance of a model |
| Net | A grouping of electrically connected components |
| Plug | A terminal of an instance |
| RoutingPad | A physical connexion (*pin*) to an instance |

## 1.2 Generalities

The C++ API has been exported in Python as closely as possible. Meaning that, save for a slight change in syntax, any code written in Python could be easily transformed into C++ code. There is no specific documentation written for the Python interface, you may directly use the C++ one.

Mostly:

- C++ namespaces are exported as Python modules.
- The *scope resolution operator* `::` converts into `.`.
- C++ blocks (between braces `{}`) are replaced by indentations.
- In C++, names are managed through a dedicated `Name` class. It has not been exported to the Python interface, you only have to use `string`.
- Coordinates are expressed in `DbU` which are `long` with a special semantic (see ??).

In `hurricane/Session.h` header we have:

```
namespace Hurricane {

   class UpdateSession {
     public:
       static void  open  ();
       static void  close ();
   };
```

```
}
```

So we can use it the following way in C++:

```
#include "hurricane/Session.h"

using namespace Hurricane;

void  doSomething ()
{
  UpdateSession::open();
  // Something...
  UpdateSession::close();
}
```

The equivalent Python code will be:

```
from Hurricane import *

def doSomething ():
    UpdateSession.open()
   # Something...
    UpdateSession.close()
```

### 1.3 Various Kinds of Constructors

Regarding the memory allocation, the Hurricane database contains two kind of objects.

1. Objects that are linked to others in the database and whose creation or deletion implies coherency operations. This is the case for Net or Horizontal. They must be created using the static **`create()`** method of their class and destroyed with their **`destroy()`** method. And, of course, they cannot be copied (the copy constructor has been disabled).

   ```
   net = Net.create( cell, 'tmp' )   # Call the static Net.create() function.
                                     # Work with this net.
                                     # ...
   net.destroy()                     # Call the dynamic destroy() method.
   ```

2. Objects that are *standalone*, like Point or Box, uses the usual construction methods. They also use the Python garbage collector mechanism and do not need to be explicitly deleted.

   ```
   def myfunc():
     bb = Box( DbU.fromLambda( 0.0)
             , DbU.fromLambda( 0.0)
             , DbU.fromLambda(15.0)
             , DbU.fromLambda(50.0) )
     return                              # bb will be freed at that point.
   ```

## 2. Setting up the Environment

### 2.1 Setting up the Paths

To simplify the tedious task of configuring your environment, a helper is provided. It will setup or modify the **`PATH`**, **`LD_LIBRARY_PATH`** (or **`DYLD_LIBRARY_PATH`** under Darwin), **`PYTHONPATH`** and **`CORIOLIS_TOP`** variables. It should automatically adapt to your kind of shell (`Bourne` like or `C-Shell` like).

```
<CORIOLIS_INSTALL>/etc/coriolis2/coriolisEnv.py
```

Use it like this (don't forget the `eval` **and** the backquotes):

```
dummy@lepka:~> eval `<CORIOLIS_INSTALL>/etc/coriolis2/coriolisEnv.py`
```

**Note**

**Do not call that script in your environment initialisation.** When used under RHEL6 or clones, it needs to be run in the **`devtoolset`** environment. The script then launches a new shell, which may cause an infinite loop if it's called again in, say **`~/.bashrc`**.
Instead you may want to create an alias:

```
alias c2r='eval "`<CORIOLIS_INSTALL>/etc/coriolis2/coriolisEnv.py`"'
```

## 2.2 User's Configurations File

You may create, in the directory you are lanching Coriolis tools, a special sub-directory `.coriolis2/` that can contain two configuration files:

- `techno.py` tells which technology to use.
- `settings.py` can override almost any default configuration setting.

Those two files are *optional*, if they do not exist the default settings will be used and the technology is `symbolic/cmos` (i.e. purely symbolic).

**Note**

Those two files will by processed by the PYTHON interpreter, so they can contain any code in addition to the mandatory variables.

### 2.2.1 The `techno.py` File

Must provide one variable named **`technology`** which values the path towards the technology file. The available technologies are installed under `<CORIOLIS_INSTALL>/etc/coriolis2`. For example, to use the 45nm FreeDPK which is in:

```
<CORIOLIS_INSTALL>/etc/coriolis2/45/freepdk_45/
```

The `techno.py` file must contain:

```
technology = '45/freepdk_45'
```

### 2.2.2 The `settings.py` File

The entries of the `parametersTable` and their definitions are detailed in CGT - The Graphical Interface.

Example of file:

```
defaultStyle = 'Alliance.Classic [black]'

parametersTable = \
    ( ('misc.catchCore'          , TypeBool      , False   )
    , ('misc.info'               , TypeBool      , False   )
    , ('misc.paranoid'           , TypeBool      , False   )
    , ('misc.bug'                , TypeBool      , False   )
    , ('misc.logMode'            , TypeBool      , False   )
    , ('misc.verboseLevel1'      , TypeBool      , False   )
    , ('misc.verboseLevel2'      , TypeBool      , True    )
    )
```

# 3. Making a Standard Cell -- Layout

In this part, we will show how to create and save a terminal Cell, that is, a cell without instances (the end point of a hierarchical design). To illustrate the case we will draw the layout of a standard cell.

We will introduce the following classes : Cell, Net, Component and its derived classes.

## 3.1 The AllianceFramework (CRL Core)

The Hurricane database only manages objects in memory. To load or save something from the outside, we need to use a *framework*. As of today, only one is available : the Alliance framework. It allows Coriolis to handle Alliance libraries and cells in the exact same way.

**Note**

To see how the AllianceFramework is configured for your installation, please have a look to `alliance.conf` in the `etc/coriolis2` directory. It must contains the same settings as the various `MBK_` variables used for ALLIANCE.

## 3.2 Session Mechanism (Hurricane)

In the Hurricane database, all modifications must take place inside an UpdateSession. At the closing of a session, created or modificated objects are fully inserted in the database. This is especially true for the visualisation, a created component will be visible *only* only after the session close.

**Note**

See `QuadTree` and `Query`.

## 3.3 Creating a new Cell (CRL Core)

The creation of a new Cell occurs through the AllianceFramework, and, as stated above, inside a UpdateSession. The AllianceFramework environment is provided by the crl module.

```
from Hurricane import *
from CRL       import *

af = AllianceFramework.get()
UpdateSession.open()

cell = af.createCell( 'my_inv' )

# Build then save the Cell.

UpdateSession.close()
```

This is the simplest call to `createCell()`, and in that case, the newly created Cell will be saved in the *working library* (usually, the current directory). You may supply a second argument telling into which library you want the Cell to be created.

In the Hurricane Cell object, there is no concept of *view*, it contains completly fused logical (netlist) and physical (layout) views.

## 3.4 The DbU Measurement Unit

All coordinates in the Hurricane database are expressed in DbU (for *Database Unit*) which are integer numbers of foundry grid. To be more precise, they are fixed points numbers expressed in hundredth of foundry grid (to allow transient non-integer computation).

To work with symbolic layout, that is, using lambda based lengths, two conversion functions are provided:

- `unit = DbU.fromLambda( lbd )` convert a lambda **`lbd`** into a `DbU`.
- `lbd = DbU.toLambda( unit )` convert a `DbU` into a lambda **`lbd`**.

In the weakly typed Python world, **`lbd`** is *float* while **`unit`** is *integer*.

## 3.5 Setting up the Abutment Box

To setup the abutment box, we use a Box which defines a box from the coordinates of the lower left corner `(x1,y1)` and upper left corner `(x2,y2)`.

```
b = Box( DbU.fromLambda( 0.0)     # x1
       , DbU.fromLambda( 0.0)     # y1
       , DbU.fromLambda(15.0)     # x2
       , DbU.fromLambda(50.0) )   # y2
cell.setAbutmentBox( b )
```

Or more simply:

```
cell.setAbutmentBox( Box( DbU.fromLambda( 0.0)
                        , DbU.fromLambda( 0.0)
                        , DbU.fromLambda(15.0)
                        , DbU.fromLambda(50.0) ) )
```

## 3.6 Adding Nets and Components

In the Hurricane terminology, a **component** is any kind of physical object among:

- Contact
- Pad
- RoutingPad
- Horizontal
- Vertical
- Plug is the only exception and will be detailed later (see ??).

Components cannot be created *alone*. They must be part of a Net.

### 3.6.1 Getting a Layer

As physical elements, Components are created using a Layer. So prior to their creation, we must get one from the database. Layers are stored in the Technology, which in turn, is stored in the DataBase. So, to get a Layer:

```
layer = DataBase.getDB().getTechnology().getLayer( 'METAL1' )
```

**Note**

**Convention for layer names.** As the database can manage both real layers and symbolic ones we adopt the following convention:

- **Real layers** are named in lowercase (`metal1`, `nwell`).
- **Symbolic layers** are named in uppercase (`METAL1`, `NWELL`).

#### 3.6.2 Creating a Net

As said above, prior to creating any Component, we must create the Net it will belong to. In that example we also make it an *external* net, that is, a part of the interface. Do not mistake the name of the net given as a string argument `'i'` and the name of the *variable* `i` holding the Net object. For the sake of clarity we try to give the variable a close name, but this is not mandatory.

```
i = Net.create( cell, 'i' )
i.setExternal( True )
```

**Note**

Unlike some other database models, in HURRICANE, **there is no explicit terminal object**, you only need to make the net external. For more information about how to connect to an external net, see 6.2 Creating Nets and connecting to Instances.

#### 3.6.3 Creating a Component

Finally, we get ready to create a Component, we will make a Vertical segment of `METAL1`.

```
segment = Vertical.create( i                              # The owner Net.
                         , layer                          # The layer.
                         , DbU.fromLambda(  5.0 )         # The X coordinate.
                         , DbU.fromLambda(  2.0 )         # The width.
                         , DbU.fromLambda( 10.0 )         # The Y source coordinate.
                         , DbU.fromLambda( 40.0 ) )       # The Y target coordinate.
```

With this overload of the `Vertical.create()` function the segment is created at an absolute position. There is a second overload for creating a relatively placed segment, see *articulated layout*.

If the net is external, that is, part of the interface of the cell, you may have to declare some of its components as physical connectors usable by the router. This is done by calling the NetExternalComponents class:

```
NetExternalComponents.setExternal( segment )
```

### 3.7 Saving to Disk (CRL Core)

Once you have finished to build your cell, you have to save it on disk. Using the AllianceFramework you can save it as a pair of file:

| View | Flag | File extension |
|---|---|---|
| Logical / Netlist | `Catalog.State.Logical` | `.vst` |
| Physical / Layout | `Catalog.State.Physical` | `.ap` |

To save both views, use the `Catalog.State.Views` flag. The files will be written in the Alliance `WORK_DIR`.

```
af.saveCell( cell, Catalog.State.Views )
```

### 3.8 The Complete Example File

The example files can be found in the `share/doc/coriolis2/examples/scripts/` directory (under the the root of the Coriolis installation).

The code needed to run it through the **`cgt`** viewer has been added. For the explanation of that part of the code, refer to 5. Make a script runnable through cgt.

```
#!/usr/bin/python

import sys
from   Hurricane import *
from   CRL       import *


def toDbU ( l ): return DbU.fromLambda(l)


def doBreak ( level, message ):
    UpdateSession.close()
    Breakpoint.stop( level, message )
    UpdateSession.open()


def buildInvertor ( editor ):
    UpdateSession.open()

    cell = AllianceFramework.get().createCell( 'invertor' )
    cell.setTerminal( True )

    cell.setAbutmentBox( Box( toDbU(0.0), toDbU(0.0), toDbU(15.0), toDbU(50.0) )

    if editor:
      UpdateSession.close()
      editor.setCell( cell )
      editor.fit()
      UpdateSession.open()

    technology = DataBase.getDB().getTechnology()
    metal1     = technology.getLayer( "METAL1"     )
    poly       = technology.getLayer( "POLY"       )
    ptrans     = technology.getLayer( "PTRANS"     )
    ntrans     = technology.getLayer( "NTRANS"     )
    pdif       = technology.getLayer( "PDIF"       )
    ndif       = technology.getLayer( "NDIF"       )
    contdifn   = technology.getLayer( "CONT_DIF_N" )
    contdifp   = technology.getLayer( "CONT_DIF_P" )
    nwell      = technology.getLayer( "NWELL"      )
    contpoly   = technology.getLayer( "CONT_POLY"  )
    ntie       = technology.getLayer( "NTIE"       )

    net = Net.create( cell, "nwell" )
    Vertical.create( net, nwell, toDbU(7.5), toDbU(15.0), toDbU(27.0), toDbU(51.

    vdd = Net.create( cell, "vdd" )
    vdd.setExternal( True )
    vdd.setGlobal  ( True )
    h = Horizontal.create(vdd, metal1, toDbU(47.0), toDbU(6.0), toDbU(0.0), toDb
    NetExternalComponents.setExternal( h )
    Contact.create ( vdd, contdifn, toDbU(10.0), toDbU(47.0), toDbU( 1.0), toDbU
    Contact.create ( vdd, contdifp, toDbU( 4.0), toDbU(45.0), toDbU( 1.0), toDbU
    Vertical.create( vdd, pdif    , toDbU( 3.5), toDbU( 4.0), toDbU(28.0), toDbU
    Vertical.create( vdd, ntie    , toDbU(10.0), toDbU( 3.0), toDbU(43.0), toDbU
```

```
        doBreak( 1, 'Done building vdd.' )

        vss = Net.create( cell, "vss" )
        vss.setExternal( True )
        vss.setGlobal  ( True )
        h = Horizontal.create(vss, metal1, toDbU(3.0), toDbU(6.0), toDbU(0.0), toDbU
        NetExternalComponents.setExternal( h )
        Vertical.create( vss, ndif    , toDbU(3.5), toDbU(4.0), toDbU(4.0), toDbU(12
        Contact.create ( vss, contdifn, toDbU(4.0), toDbU(5.0), toDbU(1.0), toDbU( 1
        doBreak( 1, 'Done building vss.' )

        i = Net.create( cell, "i" )
        i.setExternal( True )
        v = Vertical.create ( i, metal1, toDbU(5.0), toDbU(2.0), toDbU(10.0), toDbU(
        NetExternalComponents.setExternal( v )
        Vertical.create  ( i, ptrans  , toDbU( 7.0), toDbU( 1.0), toDbU(26.0), toDbU
        Vertical.create  ( i, ntrans  , toDbU( 7.0), toDbU( 1.0), toDbU( 6.0), toDbU
        Vertical.create  ( i, poly    , toDbU( 7.0), toDbU( 1.0), toDbU(14.0), toDbU
        Horizontal.create( i, poly    , toDbU(20.0), toDbU( 3.0), toDbU( 4.0), toDbU
        Contact.create   ( i, contpoly, toDbU( 5.0), toDbU(20.0), toDbU( 1.0), toDbU
        doBreak( 1, 'Done building i.' )

        nq = Net.create ( cell, "nq" )
        nq.setExternal( True )
        v = Vertical.create( nq, metal1, toDbU(10.0), toDbU(2.0), toDbU(10.0), toDbU
        NetExternalComponents.setExternal( v )
        Vertical.create( nq, pdif    , toDbU(10.0), toDbU( 3.0), toDbU(28.0), toDbU
        Vertical.create( nq, ndif    , toDbU(10.0), toDbU( 3.0), toDbU( 8.0), toDbU
        Contact.create ( nq, contdifp, toDbU(10.0), toDbU(35.0), toDbU( 1.0), toDbU
        Contact.create ( nq, contdifp, toDbU(10.0), toDbU(30.5), toDbU( 1.0), toDbU
        Contact.create ( nq, contdifn, toDbU(10.0), toDbU(10.0), toDbU( 1.0), toDbU
        doBreak( 1, 'Done building q.' )

        UpdateSession.close()
        AllianceFramework.get().saveCell( cell, Catalog.State.Views )

        return


    def scriptMain ( **kw ):
        editor = None
        if kw.has_key('editor') and kw['editor']:
          editor = kw['editor']

        buildInvertor( editor )
        return True
```

## 4. Manipulating Cells, Nets and Components

In this part, we will show how to navigate through the Nets and Components of a Cell.

### 4.1 Hurricane Collections

In Hurricane all kind of set of objects, whether organized in a real container like a `map<>` (dictionary / `dict`) or a `vector<>` (table / `list`) or an algorithmic walkthrough of the database can

be accessed through a Collection.

C++ Collections objects are exposed in Python through the *iterable* protocol, allowing to simply write:

```
for net in cell.getNets():
  print 'Components of', net
  for component in net.getComponents():
    print '|', component
```

In C++ we would have written:

```
for ( Net* net : cell->getNets() ) {
  cout << "Components of " << net << endl;
  for ( Component* component : net->getComponents() ) {
    cout << "| " << component << endl;
  }
}
```

#### 4.1.1 Restrictions about using Collections

**Never delete or create an element while you are iterating over a Collection.**

Results can be unpredictable, you may just end up with a core dump, but more subtly, some element of the Collection may be skipped or processed twice. If you want to create or delete an element, do it outside the collection loop. For example:

```
cellNets = []
for net in cell.getNets():
  cellNets.append( net )

# Remove all the anonymous nets.
for net in cellNets:
   if net.getName().endswith('nymous_'):
     print 'Destroy', net
     net.destroy()
```

### 4.2 Loading a Cell with AllianceFramework

As presented in 2.1 The Alliance Framework, the Cell that will be returned by the `getCell()` call wil be:

1. If a Cell of that name is already loaded into memory, it will be returned.

**Note**

It means that it shadows any modifications that could have been on disk since it was first loaded. Conversely, if the Cell has been modified in memory, you will get those modifications.

2. Search, in the ordered list of libraries, the first Cell that matches the requested name.

**Note**

It means that if cells with the same name exist in different libraries, only the one in the first library will be ever used. Be also aware that cell files that may remain in the `WORK_LIB`, may unexpectedly shadow cells from the libraries.

```
cell = af.getCell( 'inv_x1', Catalog.State.Views )
```

## 5. Make a script runnable through `cgt`

To use your script you may run it directly like any other Python script. But, for debugging purpose it may be helpful to run it through the interactive layout viewer **`cgt`**.

For **`cgt`** to be able to run your script, you must add to your script file a function named **`scriptMain()`**, which takes a dictionary as sole argument (**`**kw`**). The `kw` dictionary contains, in particular, the CellViewer object we are running under with the keyword `editor`. You can then load your cell into the viewer using the menu:

- `Tools` → `Python Script`. The script file name must be given without the `.py` extension.

**Note**

If you use breakpoints and want to see the progress of your script in the viewer, do not use the `--script` option of **`cgt`**.

```
ego@home:~> cgt -V --script=invertor
```

Because the script is run **before** the viewer is launched, you will only see the end result of your script.

```
def buildInvertor ( editor ):
    UpdateSession.open()

    cell = AllianceFramework.get().createCell( 'invertor' )
    cell.setTerminal( True )

    cell.setAbutmentBox( Box( toDbU(0.0), toDbU(0.0), toDbU(15.0), toDbU(50.0) )

    if editor:
      UpdateSession.close()
      editor.setCell( cell )
      editor.fit()
      UpdateSession.open()

    # The rest of the script...

    return


def scriptMain ( **kw ):
    editor = None
    if kw.has_key('editor') and kw['editor']:
      editor = kw['editor']

    buildInvertor( editor )
    return True
```

### 5.1 Using Breakpoints

It is possible to add breakpoints inside a script by calling the `Breakpoint.stop()` function. To be able to see exactly what has just been mofied, we must close the UpdateSession just before calling the breakpoint and reopen it just after. The `Breakpoint.stop()` function takes two arguments:

1. The `level` above which it will be active.
2. An informative message about the purpose of the breakpoint.

We can create a little function to ease the work:

```
def doBreak ( level, message ):
    UpdateSession.close()
    Breakpoint.stop( level, message )
    UpdateSession.open()
```

## 6. Making a hierarchical Cell -- Netlist

To illustrate the topic, we will build the netlist of a fulladder from standard cell.

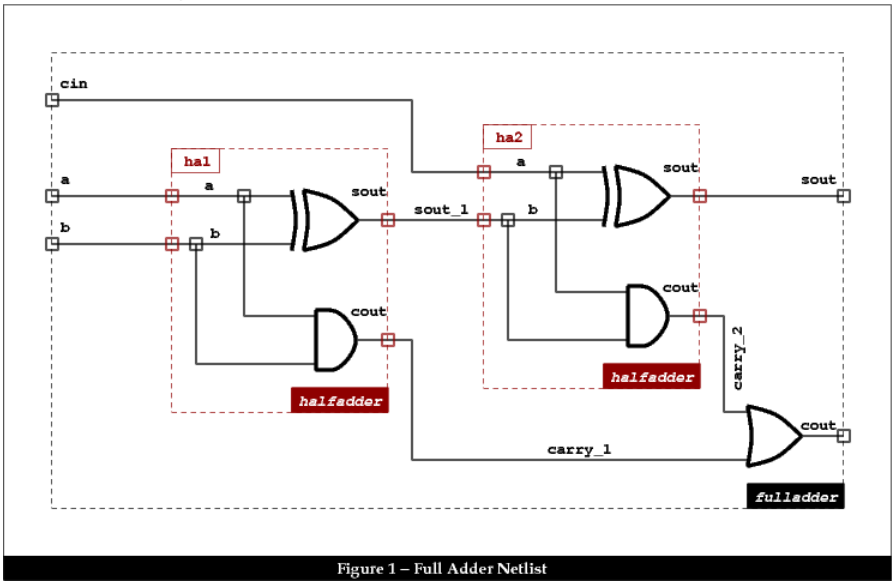


Figure 1 – Full Adder Netlist

### 6.1 Creating an Instance

Creating an Instance is straigthforward, the constructor needs only three parameters:

1. The Cell **into which** the instance is to be created.
2. The name of the instance.
3. The **master cell**, the Cell model it refers to. The master cell will be part of the hierarchical level just below the `fulladder` cell.

**Note**

Do not confuse the cell the instance is create into (`fulladder`) and the cells it refers to (the *master cell* `xr2_x2`).

```
af      = AllianceFramework.get()
xr2_x2 = af.getCell( 'xr2_x1', Catalog.State.Views )

fulladder = af.createCell( 'fulladder' )
xr2_1     = Instance.create( fulladder, 'xr2_1', xr2_x2 )
```

### 6.2 Creating Nets and connecting to Instances

An Instance has one Plug for each external net of the *master cell*. The plug allows to create a **logical** connection bewteen a Net of `fulladder` and a net from an Instance *master cell*.

A plug is somewhat equivalent to an *instance terminal* in other well known databases. Therefore, a plug is related to two nets:

1. The net of the *master cell* it is linked to. Obviously that net cannot be changed. You can access the master net with the function `plug.getMasterNet()`.

2. The net of `fulladder` the plug is connected to. This can be set, it is how we build the netlist. To set the net, use the function `plug.setNet( net )`. If the argument is `None`, the plug is *disconnected*.

To find the plug of an instance associated to a given net in the *master cell*, use `instance.getPlug( masterNet )`. The `masterNet` argument being an object of class net (not its name).

Building the **a** net of `fulladder`:

```
a = Net.create( fulladder, "a" )
a.setExternal( True )
xr2_1.getPlug( xr2_x2.getNet("i0") ).setNet( a )
a2_1 .getPlug(  a2_x2.getNet("i0") ).setNet( a )
```

**Note**

**Limitation of Hurricane Netlists.** There is no explicit terminal object in the HURRICANE database. Plugs are generated *on the fly* for each *external net* of the instance. One important consequence is that a *net* cannot appear on the interface as two differently named terminals (sometimes referred as *weekly connected* terminals). There is a strict bijection between external nets and plugs.
While it may be restrictive, it enforces cleaner designs and make it possible for the HyperNet concept/class.

## 6.3 Power supplies special case

For supplies, it may be tedious to connect the Plugs of each cell one by one (and create a lot of unneeded objects). To avoid that, we may use **Named connections**. If a signal in `fulladder` is set to *global*, then it will be considered as connected to any signal with the *same name* and *global* in the master cell of the instances.

```
vdd = Net.create( fulladder, "vdd" )
vdd.setExternal( True )
vdd.setGlobal  ( True )  # Will be connected to all the instances.
```

## 6.4 Creating the physical view of a Cell netlist

Even if loaded in the viewer, an Instance will not be displayed until it is placed.

### 6.4.1 Transformation

To place an Instance, we apply a Transformation to the coordinate system of the *master cell*. A transformation is composed of two operations:

1. An Orientation, which can be a symmetry or a rotation (or a combination of those two). The Orientation **is applied first** to the coordinate system of the *master cell*. The complete list of Orientation and their codes are given on the Orientation documentation page.

2. A **Translation**, applied in second. Translation are represented by Points.

The transformation is a change of coordinate system, be aware that if the abutment box lower left corner of the *master* cell is **not** at `(0,0)` the result of the Transformation may not be what you expect. To simplify the computation of the transformation of an instance, always place the lower left corner of the abutment box at `(0,0)`

### 6.4.2 Placing an Instance

Assuming that we want to place the cells of the `fulladder` into two rows, that the abutment box lower left corner is at `(0,0)` (same for the `xr2_x2` *master cell* layout). Here is the code to place the `xr2_1` instance to left of the second row.

Setting the translation on an Instance is not enough to make it be displayed, we also must set its *placement status* to `Instance.PlacementStatus.PLACED`.

```
xr2_1.setTransformation( Transformation( DbU.fromLambda(  0.0)
                                       , DbU.fromLambda(100.0)
                                       , Transformation.Orientation.MY ) )
xr2_1.setPlacementStatus( Instance.PlacementStatus.PLACED )
```

### 6.4.3 Nets -- From Plugs to RoutingPads

As was stated before, Plugs represent a logical connection between two levels of hierarchy. To make the physical connection to the *master net* in the instance, we now must create, in the `fulladder`, a special component which is a kind of *reference* to a component of the *master net* (in the master cell).

The so called *special component* is a RoutingPad.

The `RoutingPad` can be considered as an equivalent to `pin` in other well known databases.

```
rp = RoutingPad.create( a
                      , Occurrence( xr2_1.getPlug( xr2_x2.getNet("i0")) )
                      , RoutingPad.BiggestArea )
```

For the second parameter, we must pass an Occurrence. Occurrence objects will be explained in detail later, for now, let say that we must construct the Occurrence object with one parameter : the Plug for which we want to create a physical connection.

The RoutingPad `rp` will be a component of the `a` net.

The third argument ask the constructor of the RoutingPad to select in the master net, the component which has the biggest area.

**Note**

**Component selection.** Not all the components of a net can be selected for connection through a RoutingPad. The candidates must have been flagged with the NetExternalComponents class.
See 3.6.3 Creating a Component.

### 6.4.4 Nets -- Regular wiring

After the creation of the RoutingPads, the wiring is to be created with ordinary layout components (Horizontal, Vertical and Contact possibly articulated). Here is the complete code for net `a`. We made an articulated layout where contacts are created over RoutingPads then segments over contacts.

```
# Build wiring for a.
# Create RoutingPads first.
 rp1      = RoutingPad.create( a
                             , Occurrence( xr2_1.getPlug( xr2_x2.getNet("i0")) )
                             , RoutingPad.BiggestArea )
 rp2      = RoutingPad.create( a
                             , Occurrence(  a2_1.getPlug(  a2_x2.getNet("i0")) )
                             , RoutingPad.BiggestArea )

# Then regular wiring.
```

```
contact1 = Contact.create( rp1, via12, toDbU( 0.0), toDbU(-15.0) )
contact2 = Contact.create( rp2, via12, toDbU( 0.0), toDbU( 10.0) )
turn     = Contact.create( a  , via23, toDbU(10.0), toDbU( 35.0) )
Horizontal.create( contact2, turn     , metal2, toDbU(35.0), toDbU(2.0) )
Vertical  .create( turn    , contact1 , metal3, toDbU(10.0), toDbU(2.0) )
```

**Note**

In order to better see the layout of the wiring only, open the `Controller` and in the `Filter` tab, uncheck `Process Terminal Cells`.

## 6.5 The Complete Example File

The example file `fulladder.py` can be found in the `share/doc/coriolis2/examples/scripts/` directory (under the the root of the Coriolis installation).

```
#!/usr/bin/python

import sys
from   Hurricane import *
from   CRL       import *


def toDbU ( l ): return DbU.fromLambda(l)


def doBreak ( level, message ):
    UpdateSession.close()
    Breakpoint.stop( level, message )
    UpdateSession.open()


def buildFulladder ( editor ):

   # Get the Framework and all the master cells.
    af     = AllianceFramework.get()
    xr2_x2 = af.getCell( 'xr2_x1', Catalog.State.Views )
    a2_x2  = af.getCell( 'a2_x2' , Catalog.State.Views )
    o2_x2  = af.getCell( 'o2_x2' , Catalog.State.Views )

    UpdateSession.open()

    fulladder = af.createCell( 'fulladder' )
    fulladder.setAbutmentBox( Box( toDbU(0.0), toDbU(0.0), toDbU(90.0), toDbU(10

    if editor:
      UpdateSession.close()
      editor.setCell( fulladder )
      editor.fit()
      UpdateSession.open()

   # Create Instances.
    a2_1  = Instance.create( fulladder,  'a2_1',  a2_x2 )
    a2_2  = Instance.create( fulladder,  'a2_2',  a2_x2 )
    xr2_1 = Instance.create( fulladder, 'xr2_1', xr2_x2 )
    xr2_2 = Instance.create( fulladder, 'xr2_2', xr2_x2 )
```

```
 o2_1  = Instance.create( fulladder,  'o2_1',  o2_x2 )

# Create Nets.
 vss = Net.create( fulladder, "vss" )
 vss.setExternal( True )
 vss.setGlobal  ( True )

 vdd = Net.create( fulladder, "vdd" )
 vdd.setExternal( True )
 vdd.setGlobal  ( True )

 cin = Net.create( fulladder, "cin" )
 cin.setExternal( True )
 xr2_2.getPlug( xr2_x2.getNet('i0') ).setNet( cin )
 a2_2 .getPlug(  a2_x2.getNet('i0') ).setNet( cin )

 a = Net.create( fulladder, 'a' )
 a.setExternal( True )
 xr2_1.getPlug( xr2_x2.getNet('i0') ).setNet( a )
 a2_1 .getPlug(  a2_x2.getNet('i0') ).setNet( a )

 b = Net.create( fulladder, 'b' )
 b.setExternal( True )
 xr2_1.getPlug( xr2_x2.getNet('i1') ).setNet( b )
 a2_1 .getPlug(  a2_x2.getNet('i1') ).setNet( b )

 sout_1 = Net.create( fulladder, 'sout_1' )
 xr2_1.getPlug( xr2_x2.getNet('q' ) ).setNet( sout_1 )
 xr2_2.getPlug( xr2_x2.getNet('i1') ).setNet( sout_1 )
 a2_2 .getPlug(  a2_x2.getNet('i1') ).setNet( sout_1 )

 carry_1 = Net.create( fulladder, 'carry_1' )
 a2_1.getPlug( a2_x2.getNet('q' ) ).setNet( carry_1 )
 o2_1.getPlug( o2_x2.getNet('i1') ).setNet( carry_1 )

 carry_2 = Net.create( fulladder, 'carry_2' )
 a2_2.getPlug( a2_x2.getNet('q' ) ).setNet( carry_2 )
 o2_1.getPlug( o2_x2.getNet('i0') ).setNet( carry_2 )

 sout = Net.create( fulladder, 'sout' )
 sout.setExternal( True )
 xr2_2.getPlug( xr2_x2.getNet('q') ).setNet( sout )

 cout = Net.create( fulladder, 'cout' )
 cout.setExternal( True )
 o2_1.getPlug( o2_x2.getNet('q') ).setNet( cout )

# Instances placement.
 a2_1.setTransformation( Transformation( toDbU(0.0)
                                       , toDbU(0.0)
                                       , Transformation.Orientation.ID ) )
 a2_1.setPlacementStatus( Instance.PlacementStatus.PLACED )
 doBreak( 1, 'Placed a2_1' )

 xr2_1.setTransformation( Transformation( toDbU(  0.0)
```

```
                                          , toDbU(100.0)
                                          , Transformation.Orientation.MY ) )
    xr2_1.setPlacementStatus( Instance.PlacementStatus.PLACED )
    doBreak( 1, 'Placed xr2_1' )

    a2_2.setTransformation( Transformation( toDbU(25.0)
                                          , toDbU( 0.0)
                                          , Transformation.Orientation.ID ) )
    a2_2.setPlacementStatus( Instance.PlacementStatus.PLACED )
    doBreak( 1, 'Placed a2_2' )

    xr2_2.setTransformation( Transformation( toDbU( 45.0)
                                          , toDbU(100.0)
                                          , Transformation.Orientation.MY ) )
    xr2_2.setPlacementStatus( Instance.PlacementStatus.PLACED )
    doBreak( 1, 'Placed xr2_2' )

    o2_1.setTransformation( Transformation( toDbU(65.0)
                                          , toDbU( 0.0)
                                          , Transformation.Orientation.ID ) )
    o2_1.setPlacementStatus( Instance.PlacementStatus.PLACED )
    doBreak( 1, 'Placed o2_1' )

  # Add filler cells.
    tie_x0    = af.getCell(    'tie_x0', Catalog.State.Views )
    rowend_x0 = af.getCell( 'rowend_x0', Catalog.State.Views )
    filler_1  = Instance.create( fulladder, 'filler_1',    tie_x0 )
    filler_2  = Instance.create( fulladder, 'filler_2', rowend_x0 )

    filler_1.setTransformation( Transformation( toDbU(50.0)
                                              , toDbU( 0.0)
                                              , Transformation.Orientation.ID ) )
    filler_1.setPlacementStatus( Instance.PlacementStatus.PLACED )

    filler_2.setTransformation( Transformation( toDbU(60.0)
                                              , toDbU( 0.0)
                                              , Transformation.Orientation.ID ) )
    filler_2.setPlacementStatus( Instance.PlacementStatus.PLACED )
    doBreak( 1, 'Filler cell placeds' )

  # Getting the layers.
    technology = DataBase.getDB().getTechnology()
    metal2     = technology.getLayer( "METAL2" )
    metal3     = technology.getLayer( "METAL3" )
    via12      = technology.getLayer( "VIA12"  )
    via23      = technology.getLayer( "VIA23"  )

  # Build wiring for a.
  # Create RoutingPads first.
    rp1      = RoutingPad.create( a
                                , Occurrence( xr2_1.getPlug( xr2_x2.getNet("i0")
                                , RoutingPad.BiggestArea )
    rp2      = RoutingPad.create( a
                                , Occurrence(  a2_1.getPlug(  a2_x2.getNet("i0")
                                , RoutingPad.BiggestArea )
```

```
    # Then regular wiring.
    contact1 = Contact.create( rp1, via12, toDbU( 0.0), toDbU(-15.0) )
    contact2 = Contact.create( rp2, via12, toDbU( 0.0), toDbU( 10.0) )
    turn     = Contact.create( a  , via23, toDbU(10.0), toDbU( 35.0) )
    Horizontal.create( contact2, turn     , metal2, toDbU(35.0), toDbU(2.0) )
    Vertical  .create( turn    , contact1 , metal3, toDbU(10.0), toDbU(2.0) )

    UpdateSession.close()

    af.saveCell( fulladder, Catalog.State.Views )
    return


def scriptMain ( **kw ):
    editor = None
    if kw.has_key('editor') and kw['editor']:
      editor = kw['editor']

    buildFulladder( editor )
    return True
```

## 7. Working in real mode

The AllianceFramework only manages *symbolic* layout as Alliance does. But Coriolis is also able to work directly in *real* mode, meaning that distances will be expressed in microns instead of lambdas.

The *real* mode will be illustrated by working with the FreePDK45.

We will assume that the FreePDK45 archives is installed under:

```
/home/dks/
```

### 7.1 Loading a LEF file

Importing a lef file is simple, you just call the static function `LefImport.load()`. Multiple lef file can be imported one after another.

```
# You must set "DKsdir" to where you did install the NCSU FreePDK 45nm DK.
DKsdir  = '/home/dks'

library = LefImport.load( DKsdir + '/FreePDK45/osu_soc/lib/files/gscl45nm.lef' )
```

**Note**

**Technology checking.** The first imported LEF file must contain the technology. The technology described in the LEF file will be checked against the one configured in the running instance of CORIOLIS to look for any discrepencies.

### 7.2 Loading a BLIF file -- YOSYS

The blif format is generated by the Yosys logic synthetizer. Here again, it is pretty straightforward: call the static function `Blif.load()`. If you made your synthesis on a cell library not managed by AllianceFramework, for example the one of the FreePDK45, you must load it prior to calling the blif loader.

```
cell = Blif.load( 'snx' ) # load "snx.blif" in the working directory.
```

# 8. Tool Engines (CRL Core)

The ToolEngine class is the base class for all tools developped in Coriolis. In the rest of the tutorial we will use the names `tool` or `engine` as synonyms.

## 8.1 Placer -- Etesian

To run the placer, create the Etesian engine, then call the `place()` function.

```
import Etesian

# [...]

etesian = Etesian.EtesianEngine.create(cell)
etesian.place()
```

You can configure the placer in two ways:

1. Prior to the creation of the engine, setup an abutment for the cell. The placer will fit the cells into that area. If the area is too small, it will issue an error.
2. Setup Etesian parameters through the `settings.py` configuration file. For example:

```
parametersTable = \
    ( ("etesian.effort"           , TypeEnumerate , 2       )
    , ('etesian.uniformDensity'   , TypeBool      , True    )
    , ('etesian.spaceMargin'      , TypePercentage, 3.0     )
    , ('etesian.aspectRatio'      , TypePercentage, 100.0   )
    )
```

With this setup, the cells will be spread uniformally over the area (`etesian.uniformDensity`), with `3.0%` of free space added and an aspect ratio of `100%` (square shape).

## 8.1 Router -- Katana

Like for Etesian, you have to create the engine on the cell then call the sequence of functions detailed below.

**Note**

**Kite vs. Katana.** There are currently two routers in CORIOLIS, KITE is the old one and digital only. KATANA is a re-implementation with support for mixed routing (digital **and** analog). Until KATANA is fully implemented we keep both of them.

```
import Anabatic
import Katana

# [...]

katana = Katana.KatanaEngine.create(cell)
katana.digitalInit       ()
katana.runGlobalRouter   ()
katana.loadGlobalRouting( Anabatic.EngineLoadGrByNet )
katana.layerAssign       ( Anabatic.EngineNoNetLayerAssign )
katana.runNegociate      ( Katana.Flags.NoFlags )
```

## 8.2 A Complete Example

The example file `toolengines.py` can be found in the `share/doc/coriolis2/examples/scripts/` directory (under the the root of the Coriolis installation).

This script automatically places and routes the `fulladder` netlist as seen previously. The call to the ToolEngines is made inside the new function `placeAndRoute()`.

**Note**

As the `ToolEngine` take care of opening and closing UpdateSession, we do not need the wrapper function `doBreak()` around the breakpoints. We directly call Breakpoint.

**Note**

The space margin for this example is very high (`30%`), it's only because it's too small for the placer to run correctly. For normal case it is around `3%`.

```
#!/usr/bin/python

import sys
from   Hurricane import *
from   CRL       import *
import Etesian
import Anabatic
import Katana

# Everybody needs it.
af = AllianceFramework.get()


def toDbU ( l ): return DbU.fromLambda(l)


def buildFulladder ( editor ):

    # Get the Framework and all the master cells.
     xr2_x2 = af.getCell( 'xr2_x1', Catalog.State.Views )
     a2_x2  = af.getCell( 'a2_x2' , Catalog.State.Views )
     o2_x2  = af.getCell( 'o2_x2' , Catalog.State.Views )

     UpdateSession.open()

     fulladder = af.createCell( 'fulladder' )

    # Create Instances.
     a2_1  = Instance.create( fulladder,  'a2_1',  a2_x2 )
     a2_2  = Instance.create( fulladder,  'a2_2',  a2_x2 )
     xr2_1 = Instance.create( fulladder, 'xr2_1', xr2_x2 )
     xr2_2 = Instance.create( fulladder, 'xr2_2', xr2_x2 )
     o2_1  = Instance.create( fulladder,  'o2_1',  o2_x2 )

    # Create Nets.
     vss = Net.create( fulladder, "vss" )
     vss.setExternal( True )
     vss.setGlobal  ( True )

     vdd = Net.create( fulladder, "vdd" )
```

```
        vdd.setExternal( True )
        vdd.setGlobal  ( True )

        cin = Net.create( fulladder, "cin" )
        cin.setExternal( True )
        xr2_2.getPlug( xr2_x2.getNet('i0') ).setNet( cin )
        a2_2 .getPlug(  a2_x2.getNet('i0') ).setNet( cin )

        a = Net.create( fulladder, 'a' )
        a.setExternal( True )
        xr2_1.getPlug( xr2_x2.getNet('i0') ).setNet( a )
        a2_1 .getPlug(  a2_x2.getNet('i0') ).setNet( a )

        b = Net.create( fulladder, 'b' )
        b.setExternal( True )
        xr2_1.getPlug( xr2_x2.getNet('i1') ).setNet( b )
        a2_1 .getPlug(  a2_x2.getNet('i1') ).setNet( b )

        sout_1 = Net.create( fulladder, 'sout_1' )
        xr2_1.getPlug( xr2_x2.getNet('q' ) ).setNet( sout_1 )
        xr2_2.getPlug( xr2_x2.getNet('i1') ).setNet( sout_1 )
        a2_2 .getPlug(  a2_x2.getNet('i1') ).setNet( sout_1 )

        carry_1 = Net.create( fulladder, 'carry_1' )
        a2_1.getPlug( a2_x2.getNet('q' ) ).setNet( carry_1 )
        o2_1.getPlug( o2_x2.getNet('i1') ).setNet( carry_1 )

        carry_2 = Net.create( fulladder, 'carry_2' )
        a2_2.getPlug( a2_x2.getNet('q' ) ).setNet( carry_2 )
        o2_1.getPlug( o2_x2.getNet('i0') ).setNet( carry_2 )

        sout = Net.create( fulladder, 'sout' )
        sout.setExternal( True )
        xr2_2.getPlug( xr2_x2.getNet('q') ).setNet( sout )

        cout = Net.create( fulladder, 'cout' )
        cout.setExternal( True )
        o2_1.getPlug( o2_x2.getNet('q') ).setNet( cout )

        UpdateSession.close()

        af.saveCell( fulladder, Catalog.State.Views )
        return fulladder


    def placeAndRoute ( editor, cell ):
        # Run the placer.
        etesian = Etesian.EtesianEngine.create(cell)
        etesian.place()

        if editor:
          editor.setCell( cell )
          editor.fit()

        Breakpoint.stop( 1, 'After placement' )
```

```
    # Run the router.
    katana = Katana.KatanaEngine.create(cell)
    katana.digitalInit       ()
    katana.runGlobalRouter   ()
    katana.loadGlobalRouting( Anabatic.EngineLoadGrByNet )
    katana.layerAssign       ( Anabatic.EngineNoNetLayerAssign )
    katana.runNegociate      ( Katana.Flags.NoFlags )

    af.saveCell( cell, Catalog.State.Views )
    return


def scriptMain ( **kw ):
    editor = None
    if kw.has_key('editor') and kw['editor']:
      editor = kw['editor']

    fulladder = buildFulladder( editor )
    placeAndRoute( editor, fulladder )
    return True
```

# 9. Advanced Topics

This is a place holder as well as a reminder to myself to write this part of the documentation.

## 9.1 Occurrence

The trans-hierarchical workhorse.

## 9.2 RoutingPads

Unlike the Plugs that only make connections between two **adjacent** hierarchical levels, RoutingPads can refer to a deeply buried terminal.

## 9.3 HyperNets

This class is part of the *virtual flattening* mechanism, it allows to go through all the components of a trans-hierarchical net.

## 9.4 Miscellaeous trans-hierarchical functions

For a starter, how to get all the leaf cells...

## 9.5 Dynamically decorating data-base objects

When writing algorithms directly in Python, it may come in handy to be able to add attributes over the Hurricane data-base objects. As C++ objects exposed to the Python realm cannot natively do so (it would means to be able to modify a C++ aobject attributes *at runtime*), we add a special Property tasked with handling the extra Python attributes. The syntax has been made as simple as possible.

```
from Hurricane import PythonAttributes

class MyAttribute ( object ):
    count = 0
```

```
        def __init__ ( self ):
            self.value = MyAttribute.count
            print( '{} has been created'.format(self) )
            MyAttribute.count += 1

        def __del__ ( self ):
            print( '{} has been deleted'.format(self) )

        def __str__ ( self ):
            return '<MyAttribute {}>'.format(self.value)


    def demoAttributes ( cell ):
        PythonAttributes.enable( cell )
        cell.myAttribute0 = MyAttribute()
        cell.myAttribute1 = MyAttribute()
        print( 'cell.myAttribute0 =', cell.myAttribute0 )
        del cell.myAttribute0
        PythonAttributes.disable( cell )
```

Detailing the life cycle of Python attributes on a DBo:

1. Enabling the addition of Python attribute on a DBo:

   ```
   PythonAttributes.enable( cell )
   ```

2. Adding/removing properties on the DBo:

   ```
   cell.myAttribute0 = MyAttribute()
   cell.myAttribute1 = MyAttribute()
   print( 'cell.myAttribute0 =', cell.myAttribute0 )
   del cell.myAttribute0
   ```

3. And finally disabling the use of Python attributes on the DBo. Any still attached Python attributes will be released.

   ```
   PythonAttributes.disable( cell )
   ```

**Note**

When the attributes of a DBo are released it does not automatically imply that they are removed. Their reference count is decreased, and if they are only referenced here, they will be deleted. But if other variables still holds reference onto them, they will stay allocateds.

4. There is no need to keep track of all the DBo that have Python attributes to disable them. One can directly call:

   ```
   PythonAttributes.disableAll()
   ```