

# Tutorial 6: Graph, Blocks & Connecting

Dawei Shen\*

June 7, 2005

## Abstract

In tutorial 5, we have seen the family of flow graph classes brings great convenience and flexibility to the high level signal processing scheduling and organizing. Basically a flow graph accommodates several signal sources, sinks and signal processing blocks then connects them together to form a complete application. In this article, we aims at investigating more subtleties in a deeper level.

---

## 1 Overview

In tutorial 5, we have seen the family of ‘flow graph’ classes brings great convenience and flexibility to the high level signal processing scheduling and organizing. Basically a ‘flow graph’ accommodates several signal sources, sinks and signal processing blocks, then connects them together to form a complete application. In this article, we aims at investigating more subtleties in a deeper level.

Another example, ‘dial tone’, is chosen for illustration purpose. This is a very simple ‘Hello World!’ style example. But it should be enough to demonstrate the strength and beauty of the graph mechanism in GNU Radio. In this example, we simply generate two sine waves of different frequency and play the tones through the sound card. Only the signal source and sink are involved, without real signal processing.

The source code of this example is located at: ‘gnuradio-examples/python/audio/dial\_tone.py’. Please refer to appendix A to make sure we have the same version of the code. To run this example, no USRP board is required, but the sound card equipped on your computer. Simply input the command `./dial_tone.py` in your shell, you can hear clear tones from your speaker.

A few Python programming tips will also be reminded along the way.

## 2 Defining a function `build_graph()`

Two modules `gr` and `audio` are imported first. Then a function `build_graph()`, with no arguments, is defined:

```
def build_graph ():
```

`build_graph ()` is a global function within the module `dial_tone.py`, slightly different with the `__init__()` we talked about last time, which is a method belonging to the class `wfm_rx_graph`.

Next two constants are defined. Just a reminder: don’t forget the indentation!

---

\*The author is affiliated with Networking Communications and Information Processing (NCIP) Laboratory, Department of Electrical Engineering, University of Notre Dame. Welcome to send me your comments or inquiries. Email: dshen@nd.edu

```
sampling_freq = 32000
ampl = 0.1
```

We have met `sampling_freq` before when we talked about the audio decimation in last tutorial. It is the data rate of the digital signal entering the sound card. For most sound cards we use, 32kHz is a fairly good choice. The sampling rate is a parameter for the signal source `gr.sig_source_f` we will see later. Another parameter is the amplitude of the sine waves, `ampl`. We set it to 0.1v in this example.

### 3 Creating a graph

The graph comes!

```
fg = gr.flow_graph ()
```

`flow_graph` is a class defined in the module `gr.flow_graph.py`.

Here is a trick. Why could we use `gr.flow_graph` directly to refer to this class, not `gr.flow_graph.flow_graph`? The answer is in the initialization file `'__init__.py'` of the package `gr`. Let's look at its content:

```
from gnuradio_swig_python import *
from basic_flow_graph import *
from flow_graph import *
from exceptions import *
from hier_block import *
```

We know a directory must contain a `__init__.py` file to let Python treat it as a *package*. The statements in `__init__.py` will be executed first immediately after the package is imported somewhere. In this case, when `gr` is imported, all the definitions in `flow_graph` are imported to the package `gr`'s global symbol table. Here `*` represents for 'everything'. So we can directly access to the classes or functions defined in `flow_graph.py` using `gr.item` rather than `gr.flow_graph.item`.

`flow_graph` is derived from its 'father class' `basic_flow_graph`, the root of all 'GRAPH' related classes. `basic_flow_graph` describes the connections between blocks conceptually. Let's take a look at its source file located at:

```
/usr/local/python2.4/site-packages/gnuradio/gr/basic\_flow\_graph.py
```

By common sense, a graph should consist of vertices as well as the edges connecting them. These two basic elements are defined as `endpoint` class and `edge` class in `basic_flow_graph.py`.

Part of the definition for `endpoint` is

```
class endpoint (object):
    __slots__ = ['block', 'port']
    def __init__ (self, block, port):
        self.block = block
        self.port = port
```

In Python, `'__slot__'` is a special variable of a class. By default, all instances of classes have a 'dictionary' for attribute storage: the variable `'__dict__'`, the data type of which is `'Dictionary'`. This mechanism wastes space for objects having very few variables. The space consumption can become acute when creating large numbers of instances. The `__slots__` declaration takes a sequence of instance variables and reserves just enough space in each instance to hold a value for each variable. Space is saved because `__dict__` is not created for each instance.

So from `__slot__`, we can see the `endpoint` class has two attributes: `block` and `port`. This 2-tuple can fully specify an end point (vertex) in a graph. A `block` can be a source, a sink, or a signal process block. One `block` may have multiple ports, corresponding to different vertices in the graph.

This is part of the definition for the class `edge`:

```
class edge (object):
    __slots__ = ['src', 'dst']
    def __init__ (self, src_endpoint, dst_endpoint):
        self.src = src_endpoint
        self.dst = dst_endpoint
```

An `edge` provides a directed connection between two end points. Obviously, `edge` has two attributes: `src` and `dst`, indicating where the signal flow starts and ends. The data type for `src` and `dst` is just the class `'endpoint'` we have defined.

Based on `endpoint` and `edge`, we define the class `basic_flow_graph`. Part of its definition is:

```
class basic_flow_graph (object):
    '''basic_flow_graph -- describe connections between blocks'''
    __slots__ = ['edge_list']
    def __init__ (self):
        self.edge_list = []
```

`basic_flow_graph` has only one attribute `'edge_list'`, which is a `List` saving all the edges in the graph. It is initialized to be empty when a flow graph instance is created. The `'list'` will be modified only if we call the methods defined in the class `basic_flow_graph`, such as `'connect'` or `'disconnect'`. We will talk about these methods later.

The class `flow_graph` used in our example is derived from `basic_flow_graph`. It is defined in

[/usr/local/python2.4/site-packages/gnuradio/gr/flow\\_graph.py](/usr/local/python2.4/site-packages/gnuradio/gr/flow_graph.py)

This is part of its definition:

```
class flow_graph (basic_flow_graph):
    """add physical connection info to simple_flow_graph
    """
    __slots__ = ['blocks', 'scheduler']

    def __init__ (self):
        basic_flow_graph.__init__ (self);
        self.blocks = None
        self.scheduler = None
```

The comment reveals everything. Compared with its 'father class' `basic_flow_graph`, `flow_graph` adds the physical connection to it. `basic_flow_graph` only defines a flow graph 'abstractly', preserving the information about what kind of 'end points' have been used in the graph and which ones of them are connected by the directed 'edges'. You can think of it as a diagram drawn on your draft paper, nothing in the real world occurs. Of course neither can you run the graph. In `flow_graph`, we connect the blocks 'physically', i.e., all blocks' inputs are connected to their upstream buffers – real buffers in the computer memory. To understand the 'physical connection' more thoroughly, we can look at the following methods defined in the class `flow_graph`:

```
def _setup_connections (self):
    """given the basic flow graph, setup all the physical connections"""
    self.validate ()
    self.blocks = self.all_blocks ()
    self._assign_details ()
    self._assign_buffers ()
    self._connect_inputs ()

def _assign_details (self):
    ...
def _assign_buffers (self):
```

```

        """determine the buffer sizes to use, allocate them and attach to detail"""
        ...
    def _connect_inputs (self):
        """connect all block inputs to appropriate upstream buffers"""
        ...

```

When a graph instance calls its `start()` method to run the program, the graph will first call its `_setup_connections()` method to setup all the physical connections. It actually calls three methods: `_assign_details()`, `_assign_buffers()` and `_connect_inputs()` sequentially. The three methods finish the real connection job on a ‘physical’ level. The required buffer size is calculated and then allocated. Finally all the blocks’ input are connected to appropriate upstream buffers. The implementation details in these methods are overdetailed for us and seldom needed when we do programming with GNU Radio. We never call these methods explicitly when we play with the graph. However, I believe knowing some of the story hidden behind could help us obtain a better understanding about the operating mechanism of GNU Radio.

`flow_graph` adds two more attributes: ‘`blocks`’ and ‘`scheduler`’. `blocks` is a **List** variable, keeping the list of all blocks existing in the graph. It uses the method `all_blocks()` defined in `basic_flow_graph` to get the list. Two endpoints with the same block but different ports will be counted only once. `scheduler` is an important variable controlling the running of the flow graph. Its data type is the class `scheduler`, defined in the module `gr.scheduler`. `gr.scheduler` uses Python’s built-in module `threading`, to control the ‘starting’, ‘stopping’ or ‘waiting’ operations of the graph.

The most distinguishing parts that `flow_graph` adds to the `basic_flow_graph` are the methods that control the running of the flow graph, such as `start()`, `stop()`, `wait()` `run()`, etc. As indicated by their names, these methods are designed to actually drive the signal flows to move on or stop in the graph. `scheduler` plays a key role in these methods.

We can imagine that for most of the time when we want to create a graph instance or derive a graph class, we should use `flow_graph`, rather than `basic_flow_graph`.

OK! We have got so many stories from this single line: `fg = gr.flow_graph ()`. I hope you haven’t forgotten, we have created a graph! It’s an instance of the class `flow_graph`.

One final comment, in the introduction above, we mentioned several Python’s new data types, such as ‘**List**’ ‘**tuple**’ and ‘**Dictionary**’. They are actually part of the reason why Python is so powerful, flexible and efficient. But unfortunately, we won’t cover all these details in this tutorial. Please refer to the Python tutorial to gather more information about them.

## 4 Signal sources and sinks

### 4.1 The sources

Let’s move on. Next we create two data sources. In last tutorial, we have seen one type of source: *usrp*. It takes the USRP board as the signal input. In this example, the source is much simpler: sine waves generated by the computer:

```

src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)

```

Strictly speaking, this signal source also belongs to the category of ‘**blocks**’. As mentioned before, in GNU Radio, all the blocks are implemented using C++. The **SWIG** provides the interfaces between Python and C++, so that in Python you can directly use the class definitions or functions implemented using C++. We will talk about how to write blocks using C++ and how to use **SWIG** to construct the interfaces in tutorial 10. That will be the best time to understand the whole story. At this point, let’s forget those annoying tricks and just use those blocks safely in Python.

The best way to learn what blocks have been implemented and bundled in GNU Radio is to look at the documentation of GNU Radio generated using Doxygen. After the installation of GNU Radio, it should be located at:

</usr/local/share/doc/gnuradio-core-2.5cvs/html/index.html>

It's also available on-line here.

Press the tag 'Class Hierarchy' and look for the class `gr_sig_source_f`. Once you find it, open the class reference for it, you can see the class hierarchy of `gr_sig_source_f`. The 'root' class is `gr_block`. All signal processing blocks are derived from `gr_block`. `gr_sync_block` is an important class derived from `gr_block`. It implements a 1:1 block with optional history and certain simplifications are made. Sources and sinks, such as `gr_sig_source_f`, are derived from `gr_sync_block`. The only thing different about them is that sources have no inputs and sinks have no outputs. All these blocks are started with the prefix 'gr\_' and are implemented using C++. There is some behind-the-scenes magic when we use **SWIG** to provide the interface between C++ and Python, so that we can access `gr_sig_source_f` from Python as `gr.sig_source_f()`. The suffix '\_f' here, indicates the data type of the source, which is 'float' in our example. All these magics will be touched again when we talk about writing blocks in C++, so don't worry if you have any confusion at this point.

Anyway, the bottom line is, we have created two source instances returned by `gr.sig_source_f()`. They can be drawn into the 'graph' we created just now!

## 4.2 The sinks

We have seen the same sink before in the FM receiver: the audio device. We need to play the tones generated using the sound card.

```
dst = audio.sink (sampling_freq)
```

The `audio` module is located at:

```
/usr/local/lib/python2.4/site-packages/gnuradio/audio.py
```

The module `audio` is quite similar as `usrp`. It provides a wrapper for the audio sinks and sources. Again, there are some delicate magics behind the scene, so that you can create an audio sink instance in Python directly using `audio.sink()`. 'sink()' and 'source()' are actually functions defined in the module `audio`, which are connected to the OSS or ALSA supports for the sound cards, assuming you have installed `gr-audio-oss` or `gr-audio-alsa` modules. These two methods will return a class instance as the audio sink or source.

The story hidden from your view may be complicated, but constructing an audio sink in Python is really simple. We have done it! Let's forget those annoying details.

## 5 Connecting

Finally, we connect the blocks we have defined together to complete our flow graph:

```
fg.connect (src0, (dst, 0))
fg.connect (src1, (dst, 1))
```

The usage of the 'connect()' method is quite straightforward. But there are some issues worth mentioning. Let's visit the definition of the class `basic_flow_graph` again. This is the definition for the method `connect()`:

```
def connect (self, *points):
    '''connect requires two or more arguments that can be coerced to
    endpoints. If more than two arguments are provided, they are
    connected together successively.
    '''
    if len (points) < 2:
```

```

        raise ValueError, ("connect requires at least two endpoints;
        %d provided." % (len (points),))
    for i in range (1, len (points)):
        self._connect (points[i-1], points[i])

```

The ‘\*’ before the argument `points` is a special feature of Python’s function. It means the function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a **tuple**. Before the variable number of arguments, zero or more normal arguments may occur. So the `connect()` method can actually accept more than 2 arguments. If more than two blocks are provided to `connect()` as arguments, they will be connected one by one successively, on which the code is quite clear.

Another issue hidden behind is, when we connect the points, the points are first ‘coerced’ to be an ‘endpoint’ instance. It’s necessary to recast the argument first to make it consistent with the class `endpoint`. Let’s see the definition of the method `coerce_endpoint()`:

```

def coerce_endpoint (x):
    if isinstance (x, endpoint):
        return x
    elif isinstance (x, types.TupleType) and len (x) == 2:
        return endpoint (x[0], x[1])
    elif hasattr (x, 'block'):          # assume it's a block
        return endpoint (x, 0)
    elif isinstance(x, hier_block.hier_block_base):
        return endpoint (x, 0)
    else:
        raise ValueError, "Not coercible to endpoint: %s" % (x,)

```

It’s obvious that we don’t need to provide an entire `endpoint` instance as the argument to the `connect()` method (in fact doing this will make coding messy). We can simply provide a 2-tuple as the arguments, i.e. (block name, port No.). The `coerce_endpoint()` method will convert it to an `endpoint` instance. Like in our example, we refer to the end points as (`dst`, 0) and (`dst`, 1). If we only provide the block instance without the port number, the `coerce_endpoint()` method will add 0 as the default port number and return an `endpoint` instance, like the `src` in our example. It’s equivalent to say (`src`, 0). For blocks with only one port, it’s convenient to directly use the block name as the argument for `connect()`.

Note that as mentioned above, at this point, the graph is only created and connected logically, not physically. We only have a diagram on the draft paper.

Finally, the global function `build_graph()` returns the the flow graph instance `fg` we have created.

## 6 Run the program

Now it’s time to make the flow graph running.

```

if __name__ == '__main__':
    fg = build_graph ()
    fg.start ()
    raw_input ('Press Enter to quit: ')
    fg.stop ()

```

What’s the first line mean? Every module has an attribute ‘`__name__`’, to indicate the name of the current module. Many modules contain a paragraph of code like this:

```

if __name__ == '__main__':

```

```
The testing code
...
```

for testing purpose. When a module is imported somewhere, `'__name__'` will be created in the module's namespace, to save the name of the module file. So when a module is imported, the testing code will be ignored by the interpreter, because a module's name, `__name__` could never be `'__main__'`. However, if we directly run the module as an executable file, like `./dial_tone.py`, or using `python dial_tone.py` to run the script, not importing, then the module's namespace is the global namespace, Python set the `__name__` of the global namespace to `__main__` automatically. In such cases, the testing code will take effects.

`fg = build_graph()` calls the function we just defined and assign the returned flow graph instance to `fg`. The graph is then made running by the next line: `fg.start()`. The definitions for the methods `'start()'`, `'stop()'` are added in the class `flow_graph`:

```
def start (self):
    '''start graph, forking thread(s), return immediately'''
    if self.scheduler:
        raise RuntimeError, "Scheduler already running"
    self._setup_connections ()

    # cast down to gr_module_sptr
    # t = [x.block () for x in self.topological_sort (self.blocks)]
    self.scheduler = scheduler (self)
    self.scheduler.start ()

def stop (self):
    '''tells scheduler to stop and waits for it to happen'''
    if self.scheduler:
        self.scheduler.stop ()
        self.scheduler = None
```

When we start a graph, the first thing `start()` does, is to connect the graph 'physically', by calling the `self._setup_connections()` method, which we have discussed above. Then it creates an instance of the scheduler, and uses the scheduler to control the start or stop of the graph. We have talked about scheduler above. It plays a key role here. But its working principle might be overdetailed for us. We should feel safe to control the running of a flow graph using the methods `start()`, `stop()` in Python.

`raw_input()` is Python's build-in function, like `print`. It reads the user's input from the standard input device. For example `password = raw_input('Please input your password:')`, will print "please input your password:" on the screen and waits for the user's input. The input string is then saved in the variable `'password'`. In our example, the program just waits for the user's input to stop running. Otherwise it will run forever.

## 7 conclusion

In this article, we place a bulk of comments onto a very simple example. We've analyzed how the flow graph mechanism brings great convenience and flexibility to the GNU Radio programming. Some of the stories hidden from the scenes are also investigated to show how the mechanism is organized.

Some of the materials covered in this article may be more than necessary for us to do programming in GNU Radio at the Python level. But it should be helpful for us to understand the whole framework.

### APPENDIX A: The source code

```
from gnuradio import gr
from gnuradio import audio
```

```
def build_graph ():
    sampling_freq = 32000
    ampl = 0.1

    fg = gr.flow_graph ()
    src0 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 350, ampl)
    src1 = gr.sig_source_f (sampling_freq, gr.GR_SIN_WAVE, 440, ampl)
    dst = audio.sink (sampling_freq)
    fg.connect (src0, (dst, 0))
    fg.connect (src1, (dst, 1))

    return fg

if __name__ == '__main__':
    fg = build_graph ()
    fg.start ()
    raw_input ('Press Enter to quit: ')
    fg.stop ()
```

## References

- [1] **Python on-line tutorials**, <http://www.python.org/doc/current/tut/>
- [2] Eric Blossom, **Exploring GNU Radio**,  
<http://www.gnu.org/software/gnuradio/doc/exploring-gnuradio.html>