

Tutorial 8: Getting Prepared for Python in GNU Radio by Reading the FM Receiver Code Line by Line – Part II

Dawei Shen*

July 13, 2005

Abstract

Let's continue our discussion on the FM example `wfm_rcv_gui.py`. The usage of the GUI tools in GNU Radio, which are built upon wxPython, will be shown. We will also introduce some useful programming tips on argument parsing. If you are interested in using or even developing the GUI tools, this article would be helpful.

1 Overview

In this article, we will complete our discussion on the FM receiver code `wfm_rcv_gui.py`. One exciting feature of GNU Radio, is it incorporates powerful GUI tools for displaying and analyzing the signal, emulating the real spectrum analyzer and the oscillograph. We will introduce the usage of the GUI tools, which are built upon wxPython. Next we will talk about how to handle the command line arguments in Python.

2 GUI tools in GNU Radio

The most intuitive and straightforward way to analyze a signal is to display it graphically, both in time domain and frequency domain. For the applications in the real world, we have the spectrum analyzer and the oscillograph to facilitate us. Fortunately, in the software radio world, we also have such nice tools, thanks to wxPython, which provides a flexible way to construct GUI tools.

2.1 The 'Spectrum Analyzer' - `fft_sink`

Let's continue to read the code:

```
if 1:
    pre_demod, fft_win1 = \
        fftsink.make_fft_sink_c (self, panel, "Pre-Demodulation",
```

*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

```

                                                    512, quad_rate)
self.connect (src, pre_demod)
vbox.Add (fft_win1, 1, wx.EXPAND)

```

This is the ‘soft spectrum analyzer’, based on fast Fourier transformation (FFT) of the digital sequence. This ‘soft spectrum analyzer’ is used as the signal sink. That’s why it is named as ‘fftsink’. It’s defined in the module `wxgui.fftsink.py`. The function `make_fft_sink_c()` serves as the public interface to create an instance of the fft sink:

```

/site-packages/gnuradio/wxgui/fftsink.py
...
def make_fft_sink_c(fg, parent, title, fft_size, input_rate, ymin=0, ymax=100):
    block = fft_sink_c(fg, parent, title=title, fft_size=fft_size,
                       sample_rate=input_rate, y_per_div=(ymax - ymin)/8, ref_level=ymax)
    return (block, block.win)

```

First, we should point out that in Python, a function could return multiple values. `make_fft_sink_c()` returns two values: `block` is an instance of the class `fft_sink_c`, defined in the same module `wxgui.fftsink.py`. Another special feature of Python needs to be emphasized: Python supports multiple inheritance. `fft_sink_c` is derived from two classes: `gr.hier_block` and `fft_sink_base`. Being a ‘son’ class of `gr.hier_block` implies that `fft_sink_c` can be treated as a normal block, which can be placed and connected in a flow graph, as the next line shows:

```

self.connect (src, pre_demod)

```

`block.win` is obviously an attribute of `block`. In the definition of the class `fft_sink_c`, we can find its data type is the class `fft_window`, a subclass of `wx.Window`, also defined in the module `wxgui.fftsink.py`. We can think of it as a window that is going to be hang up on your screen. This window `block.win` will be used as the argument of the method `vbox.Add`.

2.2 How wxPython plays its role

To understand the other parts thoroughly, we need to know a little bit about wxPython, a GUI toolkit for Python. Interested readers may visit wxPython’s [website](#) or [tutorials’ page](#) for more information. It might be time consuming to explore all the technical details about wxPython. The good news is in GNU Radio, we can use the `spectrum analyzer` and `oscilloscope` pretty much as it is. Just copy those lines anywhere you want with only a few changes.

Let’s invest some time in wxPython’s organism. The first thing to do is certainly to import all wxPython’s components into current workspace, like the line ‘`import wx`’ does. Every wxPython application needs to derive a class from `wx.App` and provide an `OnInit()` method for it. The system calls this method as part of its startup/initialization sequence, in `wx.App.__init().__`. The primary purpose of `OnInit()` is to create the frames, windows, etc. necessary for the program to begin operation. After defining such a class, we need to instantiate an object of this class and start the application by calling its `MainLoop()` method, whose role is to handle the events. In our FM receiver example, where is such a class defined? Let’s look at the last three lines:

```

if __name__ == '__main__':
    app = stdgui.stdapp (wfm_rx_graph, "WFM RX")
    app.MainLoop ()

```

In fact, such a class, called `stdapp` has been created when we import the `stdgui` module.

```

from gnuradio.wxgui import stdgui, fftsink

```

The final two lines again will probably be the same for all your wxPython applications. We simply create an instance of our application class, and then call its `MainLoop()` method. `MainLoop()` is the heart of the application and is where events are processed and dispatched to the various windows in the application. There are some tricks behind the scene. Don't worry about that.

Let's look at the definition of `stdapp` in `/gnuradio/wxgui/stugui.py`:

```
class stdapp (wx.App):
    def __init__ (self, flow_graph_maker, title="GNU Radio"):
        self.flow_graph_maker = flow_graph_maker
        self.title = title
        # All our initialization must come before calling wx.App.__init__.
        # OnInit is called from somewhere in the guts of __init__.
        wx.App.__init__ (self)

    def OnInit (self):
        frame = stdframe (self.flow_graph_maker, self.title)
        frame.Show (True)
        self.SetTopWindow (frame)
        return True
```

`stdapp` is right the class derived from `wx.App`. Its initialization method `__init__()` takes two arguments: `flow_graph_maker`, a class belonging to the flow graph family (remember the biggest class `wfm_rx_graph` we created is derived from `gr.flow_graph?`); `title`, the title of the whole application (WFM RX in our example). In `OnInit()` method, these two arguments are further used to create the object of `stdframe`.

```
class stdframe (wx.Frame):
    def __init__ (self, flow_graph_maker, title="GNU Radio"):
        # print "stdframe.__init__"
        wx.Frame.__init__(self, None, -1, title)

        self.CreateStatusBar (2)
        mainmenu = wx.MenuBar ()

        menu = wx.Menu ()
        item = menu.Append (200, 'E&xit', 'Exit')
        self.Bind (wx.EVT_MENU, self.OnCloseWindow, item)
        mainmenu.Append (menu, "&File")
        self.SetMenuBar (mainmenu)

        self.Bind (wx.EVT_CLOSE, self.OnCloseWindow)
        self.panel = stdpanel (self, self, flow_graph_maker)
        vbox = wx.BoxSizer(wx.VERTICAL)
        vbox.Add(self.panel, 1, wx.EXPAND)
        self.SetSizer(vbox)
        self.SetAutoLayout(True)
        vbox.Fit(self)

    def OnCloseWindow (self, event):
        self.flow_graph().stop()
        self.Destroy ()

    def flow_graph (self):
        return self.panel.fg
```

It's worth taking a while to understand the layout of a wxPython GUI. In wxPython, a `wx.Window` is anything which can take up visual space on the computer screen. Thus, the `wx.Window` class is the base class from which all visual elements are derived – including input fields, pull-down menus, buttons, etc. The `wx.Window` class defines all the behavior common to all visual GUI elements, including positioning, sizing, showing, giving focus, etc. If we're looking for an object to represent a window on the computer screen, don't look at `wx.Window`, look at `wx.Frame` instead. `wx.Frame` is derived from `wx.Window`. It implements all behavior specific to windows on the computer's screen. A 'Frame' is a window whose size and position can (usually) be changed by the user. It usually has thick borders and a title bar, and can optionally contain a menu bar, toolbar and status bar. A 'Frame' can contain any window that is not a frame or dialog. So to create a 'window' on the computer screen, you create a `wx.Frame` (or one of its sub-classes, such as `wx.Dialog`), rather than a `wx.Window`.

Within a frame, you'll use a number of `wx.Window` sub-classes to flesh out the frame's contents, such as `wx.MenuBar`, `wx.StatusBar`, `wx.ToolBar`, sub-classes of `wx.Control` (eg. `wx.Button`, `wx.StaticText`, `wx.TextCtrl`, `wx.ComboBox`, etc.), or `wx.Panel`, which is a container to hold your various `wx.Control` objects. All visual elements (`wx.Window` objects and their subclasses) can hold sub-elements. A `wx.Frame` might hold a number of `wx.Panel` objects, which in turn hold a number of `wx.Button`, `wx.StaticText` and `wx.TextCtrl` objects.

In our example, `stdframe`, the subclass of `wx.Frame`, is used to create the 'frame'. We make this frame appear by 'showing' it using `frame.Show (True)`. The `SetTopWindow()` method simply tells that this frame is one of the main frames (in this case the only one) for the application. Notice the shape of the constructor of `wx.Frame`:

```
wx.Frame(Parent, Id, "title")
```

Most of the constructors in wxPython have this shape: A parent object as a first parameter and an Id in a second parameter. As shown in the example, it's possible to use respectively `None` and `-1` as default parameters, meaning the object has no parent and respectively a system-defined Id.

In `stdframe.__init__()`, we create a `panel` and place inside the frame.

```
self.panel = stdpanel (self, self, flow_graph_maker)
```

The class `stdpanel` is derived from `wx.Panel`:

```
class stdpanel (wx.Panel):
    def __init__ (self, parent, frame, flow_graph_maker):
        wx.Panel.__init__ (self, parent, -1)
        self.frame = frame

        vbox = wx.BoxSizer (wx.VERTICAL)
        self.fg = flow_graph_maker (frame, self, vbox, sys.argv)
        self.SetSizer (vbox)
        self.SetAutoLayout (True)
        vbox.Fit (self)

        self.fg.start ()
```

Note that `panel`'s parent is the `frame` object we create just now, meaning this panel is a subcomponent of the frame. The `frame` places the `panel` inside itself using a `wx.BoxSizer`, `vbox`. The basic idea behind a box sizer is that windows will most often be laid out in rather simple basic geometry, typically in a row or a column or nested hierarchies of either. A `wx.BoxSizer` will lay out its items in a simple row or column, depending on the orientation parameter passed to the constructor. In our example, `vbox = wx.BoxSizer(wx.VERTICAL)` tells the constructor all the items will be placed vertically. The `SetSizer()` call tells the frame which sizer to use. The call to `SetAutoLayout()` tells

the frame to use the sizer to position and size your components. And finally, the call to `sizer.Fit()` tells the sizer to calculate the initial size and position for all its elements. If you are using sizers, this is the normal process you would go through to set up your window or frame's contents before it is displayed for the first time. The most important and useful method for a sizer is `add()`, it appends an item to the sizer. Its syntax looks like:

```
Add(self, item, proportion=0, flag=0)
```

'item' is what you wish to append to the sizer, usually a `wx.Window` object. It can also be a child sizer. `proportion` is related to whether a child of a sizer can change its size in the main orientation of the `wx.BoxSizer`. There are several flags defined in `wx`, and they are used to determine how the sizer items behave and the border of the window. `wx.EXPAND` used in our example means the item will be expanded to fill the space allotted to the item. Refer to [this page](#) for a complete description.

Have you ever had this confusion: we define a 'big' class `wfm_rcv_graph`, but where do we use it? why do we never create an instance of this class? The secret is revealed in `stdpanel.__init__()`. The instance of `wfm_rcv_graph` is created here and the flow graph is started.

```
class stdpanel (wx.Panel):
    def __init__ (self, parent, frame, flow_graph_maker):
        .....
        vbox = wx.BoxSizer (wx.VERTICAL)
        self.fg = flow_graph_maker (frame, self, vbox, sys.argv)
        self.SetSizer (vbox)
        self.SetAutoLayout (True)
        vbox.Fit (self)
        self.fg.start ()
```

We put a panel in the frame, but what do we put in the panel? We first create a new sizer `vbox` for the panel. Note that this `vbox` is different from the one defined in `stdframe`. Then we create an instance of `flow_graph_maker` (`wfm_rcv_graph`) with `vbox` as an argument passed to it (also with `frame` and the `panel` itself). In `wfm_rcv_graph.__init__()`, this `vbox` will append several spectrum analyzers or oscillograph (`wx.Window` objects) to the sizer by using `vbox.Add()`. Then the panel uses the sizer `vbox` position and size all these child-windows. Finally, we start the flow graph: `self.fg.start()`, the corresponding data would be displayed on the screen dynamically.

Let's go back to the code of our FM receiver example.

```
if 1:
    pre_demod, fft_win1 = \
        fftsink.make_fft_sink_c (self, panel, "Pre-Demodulation",
                                512, quad_rate)

    self.connect (src, pre_demod)
    vbox.Add (fft_win1, 1, wx.EXPAND)
```

and the definition of the `make_fft_sink_c()` method:

```
/site-packages/gnuradio/wxgui/fftsink.py
...
def make_fft_sink_c(fg, parent, title, fft_size, input_rate, ymin=0, ymax=100):
    block = fft_sink_c(fg, parent, title=title, fft_size=fft_size,
                       sample_rate=input_rate, y_per_div=(ymax - ymin)/8, ref_level=ymax)
    return (block, block.win)
```

Everything is much clearer now, right? `panel` is passed to `make_fft_sink_c()` as the 'parent' of this fft sink (`block.win`, a `wx.Window` object). The return value of `block.win` is saved in `fft_win1` and then appended to `vbox`.

`make_fft_sink_c()` takes seven parameters. `fft_size` is the number of samples used to perform FFT. `input_rate` is the sample frequency. `ymin` and `ymax` give the vertical range of the plot you wish to display on the screen.

Note that there is a complicated story behind the class `fft_sink_c`. We didn't talk about how fast Fourier transform is performed and how it is used as a block, but focusing on its interface to Python and wxPython. In fact, another Python package called 'Numeric' helps a lot here. However, we don't need to know all the details. Understanding how it interacts with wxPython and other blocks at the Python level would be sufficient.

2.3 The 'Oscillograph'- `scope_sink`

Another important GUI tool in GNU Radio is the 'soft oscillograph' - `scope_sink`. It's not used in our example, but it would be very helpful if you wish to see the waveforms in the time domain. Its usage is quite similar to the `fft_sink`:

```
if 1:
    scope_input, scope_win1 = \
        scopesink.make_scope_sink_f (self, panel, "Title", self.fs)
    self.connect (signal, scope_input)
    vbox.Add (scope_win1, 1, wx.EXPAND)
```

Note that here `signal` should be a real float signal. If you wish to display a complex signal with I/Q channels, `make_scope_sink_c()` is the right choice. Copy these lines wherever you think a scope should appear, then connect it to the signal as a block. Refer to `/site-packages/gnuradio/wxgui/scopesink.py` for more details.

3 Handling command line arguments

Python fully supports creating programs that can be run on the command line, complete with command-line arguments and either short- or long- style flags to specify various options. Remember when we create an instance of `wfm_rcv_graph` in `stdpanel.__init__()`, we use:

```
self.fg = flow_graph_maker (frame, self, vbox, sys.argv)
```

Each command line argument passed to the program will be saved in `sys.argv`, which is just a 'list'. In this list, `sys.argv[0]` is just the command itself (`wfm_rcv_gui.py` in our example). So actually all the arguments are saved in `sys.argv[1:]`. That explains why we use `IF_freq = parseargs(argv[1:])` to process the arguments.

You may want to use short- or long- style flags to add various options like '-v', or '--help'. Then the `optparse` module is exactly what you would like to use. `optparse` is a powerful and flexible command line interpreter. You may see [this page](#) to study it. Another example, located at

```
gnuradio-examples/python/usrp/fsk_r(t)x.py
```

gives a very good demonstration on how to use this parser.

4 conclusion

This tutorial completes our discussion on the FM receiver example. We mainly talk about how wxPython plays its role in GNU Radio. It might be a little bit involved to understand how those

classes are organized together, but it won't be that difficult if we are patient enough. Besides, the good news is we can simply use those codes as templates, without worrying too much about the implementation details.

References

- [1] **Python on-line tutorials**, <http://www.python.org/doc/current/tut/>
- [2] **Python Library Reference - optparse**,
<http://www.python.org/doc/2.3/lib/module-optparse.html>
- [3] **wxPython on-line tutorials**, <http://wxpython.org/tutorial.php>
- [4] **wxPython wiki, Getting started**, http://wiki.wxpython.org/index.cgi/Getting_20Started