Ryan's Ramblings

Why not?...

Building a Matplotlib GUI with Qt Designer: Part 2

In this second blog post, I'm going to add some of the custom logic to the application GUI that was constructed using Qt Designer in the Part 1. We'll be using Python (version 3.4), PyQt (version 4.10.4), and matplotlib (version 1.4.3), but this code should work with Python 2.7 as well. I've found that these packages are most easily installed using Continuum Analytics' Anaconda Python Distribution.

To begin, create an empty Python file called "custommpl.py" that will contain all the code in this example. This file should be located in the same directory as the UI file "window.ui" from Part 1. The final version of this Python file is included in the project zip package.

Imports

The necessary imports are the first element of our program and are shown below.

```
1 from PyQt4.uic import loadUiType
2
3 from matplotlib.figure import Figure
4 from matplotlib.backends.backend_qt4agg import (
5     FigureCanvasQTAgg as FigureCanvas,
6     NavigationToolbar2QT as NavigationToolbar)
```

Let's examine the matplotlib imports first. Notice that the generic `pyplot` module is not imported — do *not* use the functions from this module. The `pyplot` module defines wrapper functions that are meant to automate the creation of the interactive plotting window, and they will most likely cause problems for any custom GUI object. The`Figure` class imported above is a very generic plotting container for all aspects of our plot. Instances of this object have the same methods as the figure objects created using `pyplot.figure`. In addition, we've also imported `FigureCanvasQTAgg` and the `NavigationToolbar2QT`from the PyQt4 backend module. These are both custom Qt widgets. A canvas contains and displays a Figure instance, and a navigation toolbar is the widget containing the buttons for interacting with the plot.

The `loadUiType` function requires a single argument, the name of a Designer UI file, and returns a tuple of the appropriate application classes. This can be done directly after the imports, as shown below.

```
1 Ui_MainWindow, QMainWindow = loadUiType('window.ui')
```

In this case, the return tuple contains two class definitions. The first is our custom GUI application class, set to the `Ui_MainWindow` variable. The second is the base class for our custom GUI application, which in this case is `PyQt4.QtGui.QMainWindow`. The base class type is defined by the widget we first created with Designer. Note: These are *not* instances of these classes, but the class definitions themselves. They are meant to serve as superclasses to our new application logic class, which we'll create next.

This usage differs from many other examples. For tutorials typically convert the UI file to a Python module using the `pyuic4` utility (see Part 1 for details), and then import that module into a new script, i.e. `from window import Ui_MainWindow`. The downside of this approach is that, in addition to generating an extra file, others might be tempted to modify the Python module directly. If the application design is changes, which updates the UI file, then the `pyuic4` conversion will destroy any custom modifications to the Python module. By using the UI file directly, as I've done above, it will not be possible to inadvertently alter the GUI design code.

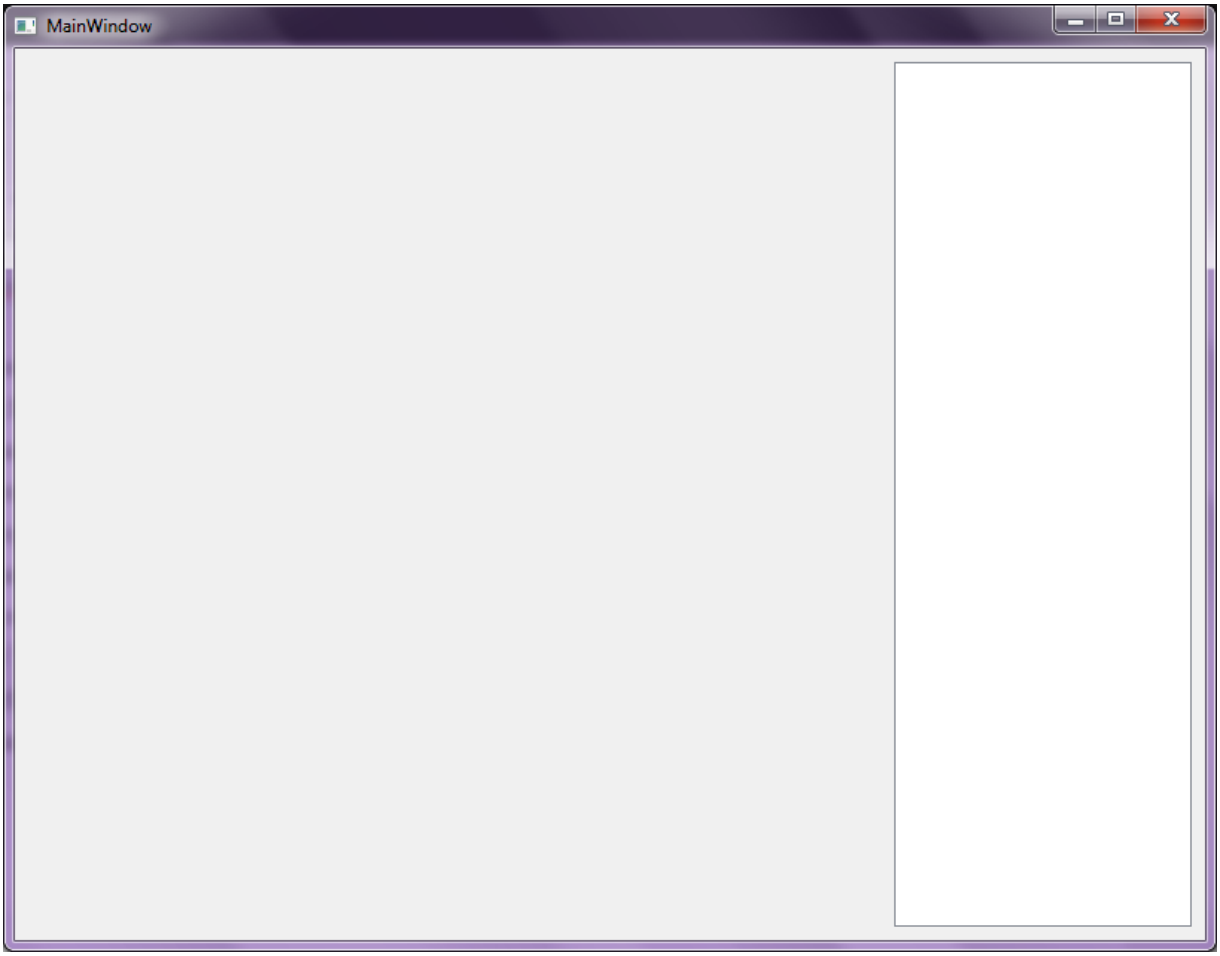
The Minimal Subclass

Now we are ready to create a minimal subclass to run our GUI application. To start, we define a new class, called `Main`, that inherits from `QMainWindow` and `Ui_MainWindow`. Initialization of this class should call the `__init__` method of `QMainWindow` as well as the `setupUi` method, which is defined in all Designer-created GUI applications.

```
1  class Main(QMainWindow, Ui_MainWindow):
2      def __init__(self, ):
3          super(Main, self).__init__()
4          self.setupUi(self)
5
6  if __name__ == '__main__':
7      import sys
8      from PyQt4 import QtGui
9
10     app = QtGui.QApplication(sys.argv)
11     main = Main()
12     main.show()
13     sys.exit(app.exec_())
```

To generate a functional GUI, a conditional code block is added to the end of the script. The leading `if` statement is telling Python to process this code block only if the program is run directly (e.g. `$ python custommpl.py`) rather than being imported into a different Python program (e.g. `import custommpl`). `QApplication` starts a Qt GUI event

loop, which is a necessary prerequisite to running any Qt GUI applications. Instantiating`Main` creates our custom application, but on its own, this will not display the GUI. This is accomplished by calling the `show` method. The `sys.exit` line simply ensures that the GUI event loop is closed properly when we quit the program. Running this file from the command line should produce the following window.



Great! We've successfully created our first GUI! But at this point, it doesn't do anything...

Adding a Plot

Next, add a new method, `addmpl`, to our `Main` class that inserts a plotting Figure instance into our matplotlib container.

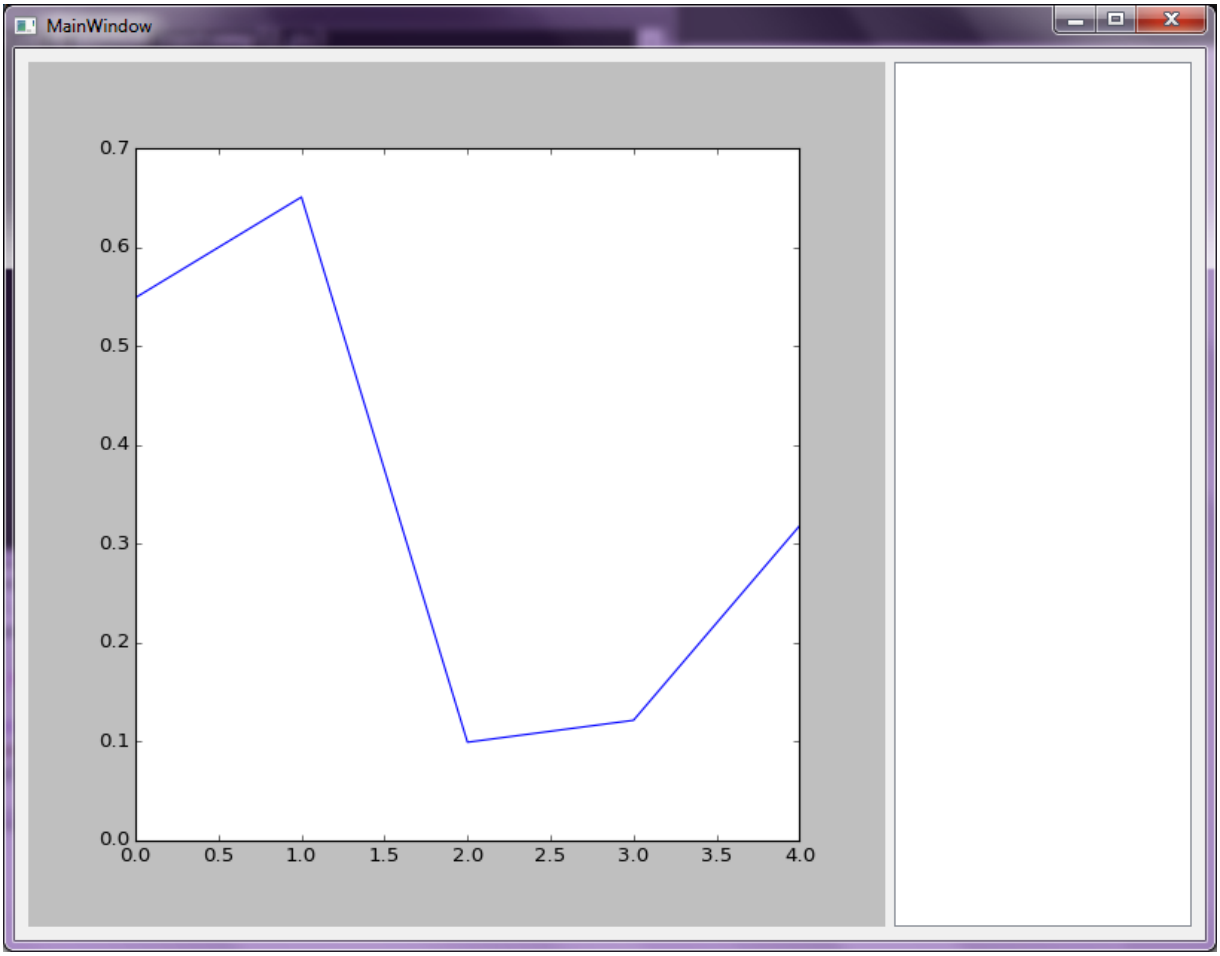
```
1 class Main(QMainWindow, Ui_MainWindow):
2     def __init__(self, ):
3         super(Main, self).__init__()
4         self.setupUi(self)
5 
6     def addmpl(self, fig):
7         self.canvas = FigureCanvas(fig)
8         self.mplvl.addWidget(self.canvas)
9         self.canvas.draw()
```

The second line creates a FigureCanvas widget instance to contain our plot. This is stored as a `Main` instance attributed called `canvas`. Next, we need to remember a little information about our matplotlib container widget, which we created in Part 1. Recall that we enforced a vertical layout for any encapsulated widgets in `mplwindow` and named this vertical layout `mplvl`. This is attribute is a `QVBoxLayout` instance, which has an `addWidget` method for adding new vertically-aligned widgets into the container widget. We'll use this method to upload our Figure widget. Calling the `draw` method on the canvas to renders the plot.

Side Note: The canvas object could have been added directly to our `mplwindow` container widget without using the vertical layout. This is done by setting the parent widget of the canvas: `self.canvas.setParent(self.mplwindow)`. However, in this case, the figure will not expand to fit the container widget, and it will not resize with the window. In addition to vertically stacking widgets, the vertical layout instance ensures that the widgets is manages are properly stretched to fit the available area.

As a test, create a simple Figure instance in the conditional code block at the end of the module. Add this plot to the GUI using our new `addmpl` method (line 12). The code and resulting plot are show below.

```
1  if __name__ == '__main__':
2      import sys
3      from PyQt4 import QtGui
4      import numpy as np
5
6      fig1 = Figure()
7      ax1f1 = fig1.add_subplot(111)
8      ax1f1.plot(np.random.rand(5))
9
10     app = QtGui.QApplication(sys.argv)
11     main = Main()
12     main.addmpl(fig1)
13     main.show()
14     sys.exit(app.exec_())
```

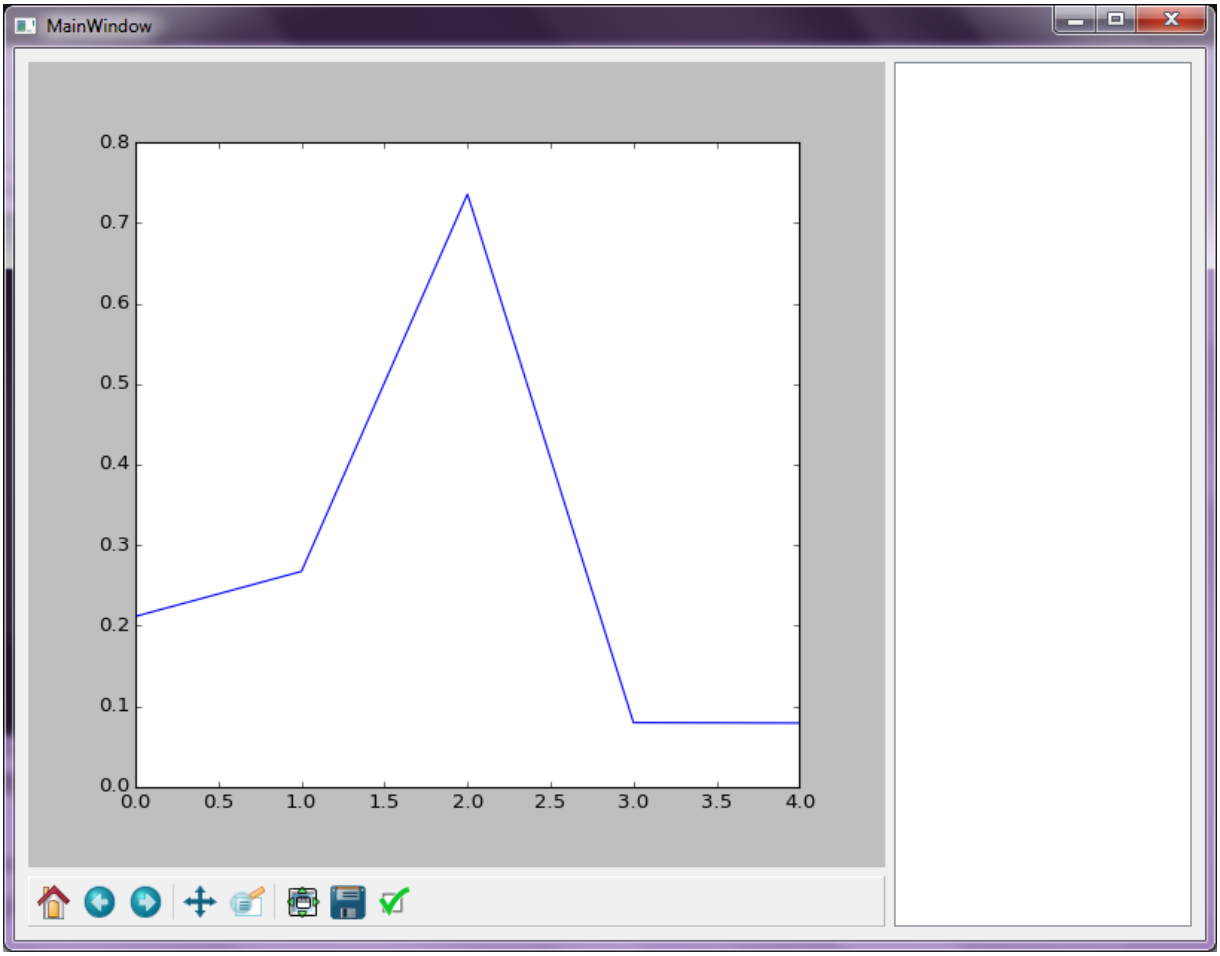


Adding a Toolbar

Although we've successfully added a plot to our application, there is currently no way to interact with the data. This is where the NavigationToolbar comes into play. Adding the toolbar to our plot involves a simple modification to the `addmpl` method.

```
1      def addmpl(self, fig):
2          self.canvas = FigureCanvas(fig)
3          self.mplvl.addWidget(self.canvas)
4          self.canvas.draw()
5          self.toolbar = NavigationToolbar(self.canvas,
6                  self.mplwindow, coordinates=True)
7          self.mplvl.addWidget(self.toolbar)
```

The `NavigationToolbar` construction should be straightforward (lines 5-6). First, you need to connect this toolbar with canvas containing the axes and data. The second argument is the parent widget that will house this toolbar, which in this case is our `mplwindow` widget. The `coordinates` keyword argument controls the display of the cursor coordinates when the mouse is hovered over an axes. Finally, the toolbar widget is added into the vertical layout widget `mplvl` in the same way as the canvas. Because the toolbar was the second widget added, it will be placed under the plot window. The final version of this application is shown below.



To try something different, we can make the toolbar a detachable widget on the main window by using the following code. Compare the code below with the first example.

```
1      def addmpl(self, fig):
2          self.canvas = FigureCanvas(fig)
3          self.mplvl.addWidget(self.canvas)
4          self.canvas.draw()
5          self.toolbar = NavigationToolbar(self.canvas,
6                  self, coordinates=True)
7          self.addToolBar(self.toolbar)
```

Changing plots

In order to change plots, we're going to do something a little different. In principle, we could use the matplotlib`axes` instances to modify the displayed data directly. However, this makes it hard to change the plot type/shape/number of the subplots. Instead, we will define a new method called `rmmpl` inside our `Main` class that completely removes the current plot and toolbar widget. New figures can then be added by calling the `addmpl` method with a new Figure instance. The advantage here is that Figure instances retain all of their plotting and layout information, so we don't have to recreate these aspect with each plot change. Below is our code.

```
1      def rmmpl(self,):
```

```
        self.mplvl.removeWidget(self.canvas)
        self.canvas.close()
        self.mplvl.removeWidget(self.toolbar)
        self.toolbar.close()
```

This should be pretty easy to understand. The `removeWidget` method removes the canvas and toolbar from our vertical layout instance. Calling the`close` method on these widgets ensures that they are no longer displayed in the application window. If we don't do this, new widgets will be overlaid on the previous plots, which causes problems.

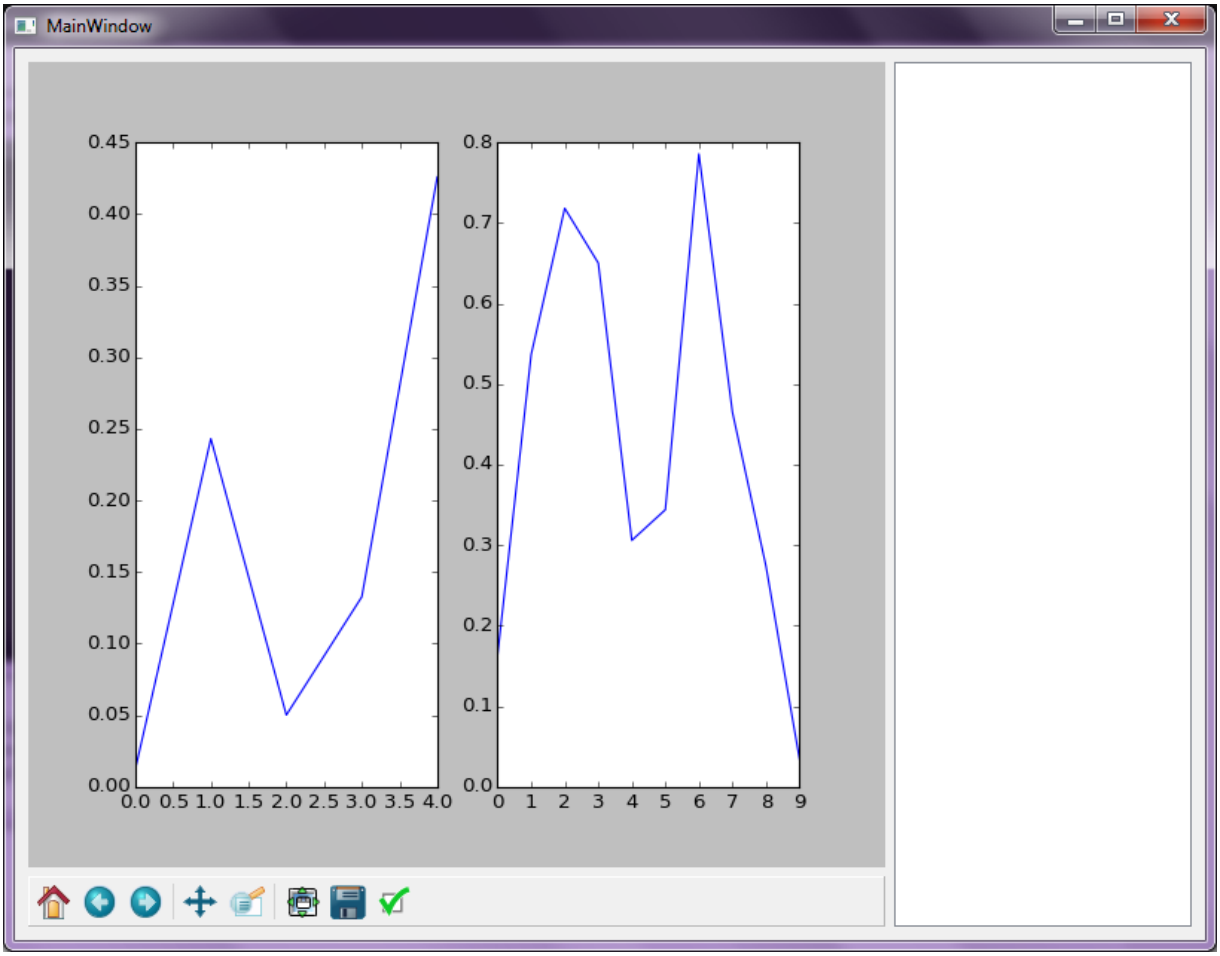
To see how this might work, change the conditional code as shown below. Hit `Enter` in the terminal to break out of the `input` and switch plots. The second plot should look something like the one shown below.

```
if __name__ == '__main__':
    import sys
    from PyQt4 import QtGui
    import numpy as np

    fig1 = Figure()
    ax1f1 = fig1.add_subplot(111)
    ax1f1.plot(np.random.rand(5))

    fig2 = Figure()
    ax1f2 = fig2.add_subplot(121)
    ax1f2.plot(np.random.rand(5))
    ax2f2 = fig2.add_subplot(122)
    ax2f2.plot(np.random.rand(10))

    app = QtGui.QApplication(sys.argv)
    main = Main()
    main.addmpl(fig1)
    main.show()
    input()
    main.rmmpl()
    main.addmpl(fig2)
    sys.exit(app.exec_())
```



Now we've created a very simple plotting window, to which we can dynamically change the displayed Figure instances, the final step in this process will be to add the figures to our list widget so users can easily switch plots by simply selecting a new different title in the list. Coding that behavior is straightforward, but will be covered in detail in new next post.

February 19, 2015 Ryan Python Matplotlib, PyQt, Python Classes

Proudly powered by WordPress