Building a Matplotlib GUI with Qt Designer: Part 3

This series of blog posts details the creation of a custom Qt application containing an interactive matplotlib widget and a plot selection list, which controls the currently displayed figure. In Part 1, we constructed our application framework and layout using Qt Designer. Part 2 of this series explored the custom subclass creation that was necessary to add custom logic to our application. In this installment, we will look at adding multiple figures to our application and including their names in the list widget. In addition, we will write some code to change the current figure by selecting from the names in the list widget. The final versions of the Qt Designer UI file and the "custommpl.py" script can be downloaded here.

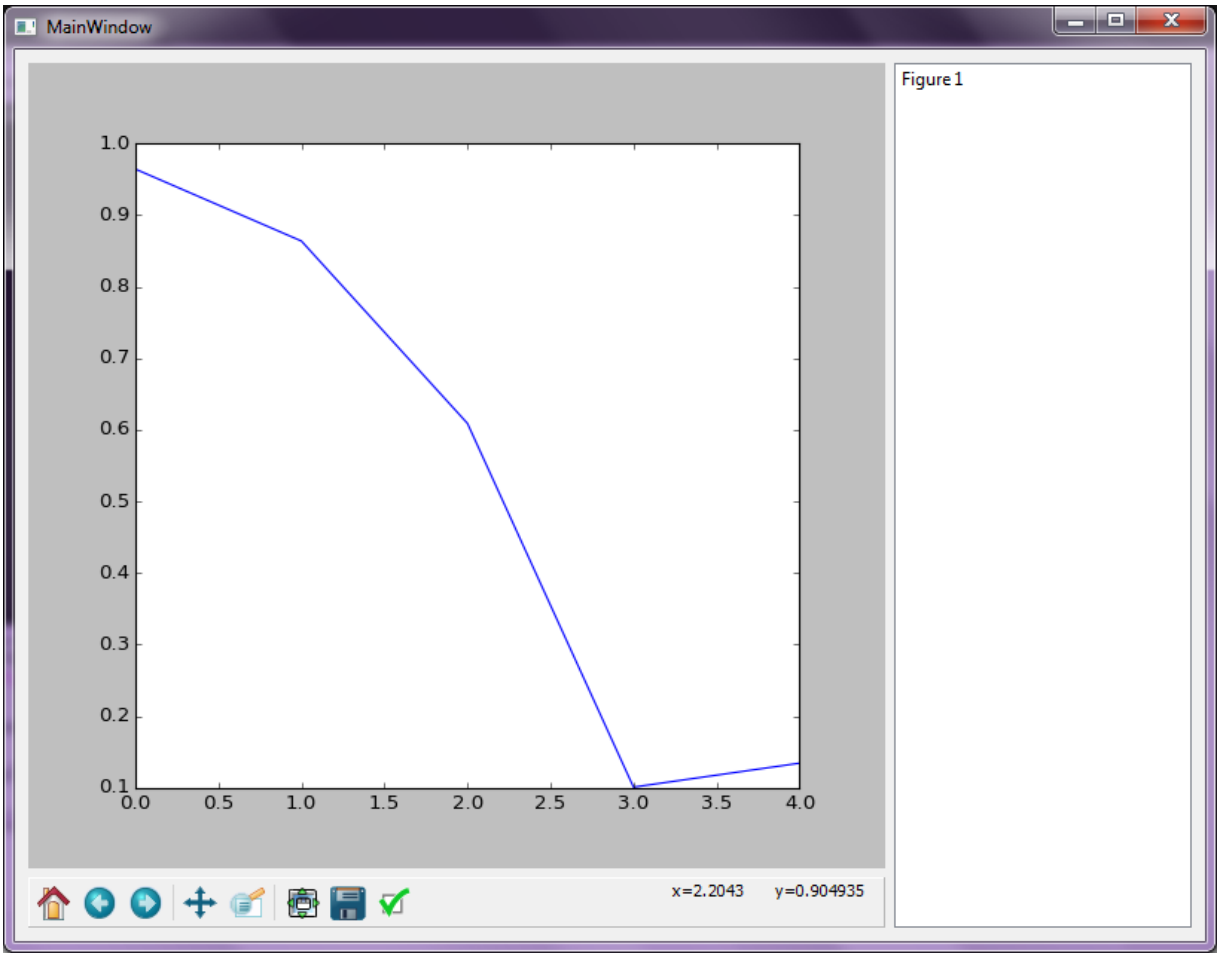
Adding Items to the List

Adding text to our list is simple. Our list widget, which we called `mplfigs`, is an instance of the `QListWidget` class. This class has an `addItem` method which adds the given string argument to the list. Remember that if we also want to be able to switch figures by selecting names from the list, so to get us started, let's modify the `Main` class's `__init__` method and add a new method called`addfig`.

```
1      def __init__(self, ):
2          super(Main, self).__init__()
3          self.setupUi(self)
4          self.fig_dict = {}
5
6      def addfig(self, name, fig):
7          self.fig_dict[name] = fig
8          self.mplfigs.addItem(name)
```

Now any new instances of our application will contain an empty dictionary called `fig_dict`, which we will use to store our Figure instances by name. The new `addfig` method takes a `name` string argument and a Figure instance. The Figure is added to the dictionary under the `name` key, which is also added to our list widget. To see how this works, change the condition code block at the end of the program as follows. The resulting application window is 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))

    app = QtGui.QApplication(sys.argv)
    main = Main()
    main.addmpl(fig1)
    main.addfig('Figure 1', fig1)
    main.show()
    sys.exit(app.exec_())
```



To add a new figure, call `rmmpl` to remove the old figure then use `addmpl` and `addfig` to add the next figure. The plot will update and the new names will be added to the list; however, our application will have no way of knowing how to change the plots when you select names from the list.

Changing Plots Using the List

To connect the names in the list to the figure updating code, we need to take advantage of Qt's signal and slot architecture. The main Qt documentation has a very nice explanation of this concept, so I'll just give a very brief overview here. Essentially, any "events" that occurs in a widget trigger a particular "signal", which is defined by the widget. In our example, when an item in our list widget is selected,

it triggers an `itemClicked` signal from the list, which emits a `QListWidgetItem` instance. Initially, the signal is not directed anywhere, so nothing happens. However, we can connect this signal to another widget's slot, which is a (potentially custom) method that handles the emitted information. In our case, when the `itemClicked` signal is triggered, we want to remove the old figure (`rmmpl`) and add the figure (`addmpl`) defined by the emitted `QListWidgetItem` name. To make this possible, we "connect" the `itemClicked` signal to a new method that we'll call `changefig`. This is more clearly seen in example code.

```
class Main(QMainWindow, Ui_MainWindow):
    def __init__(self, ):
        super(Main, self).__init__()
        self.setupUi(self)
        self.fig_dict = {}

        self.mplfigs.itemClicked.connect(self.changefig)

        fig = Figure()
        self.addmpl(fig)

    def changefig(self, item):
        text = item.text()
        self.rmmpl()
        self.addmpl(self.fig_dict[text])
```

A few changes have been made to our initialization method. First of all, we connected the `itemClicked` signal to a new method called `changefig`. Remember that when this signal is triggered it emits a `QListWidgetItem` as well, so `changefig` must accept one argument, which is an instance of the list item. List widget items define a `text` method, which simple returns the text of the selected item. Once we know the item text, we can remove the old figure and display a new one based on the name defined by the selected text.

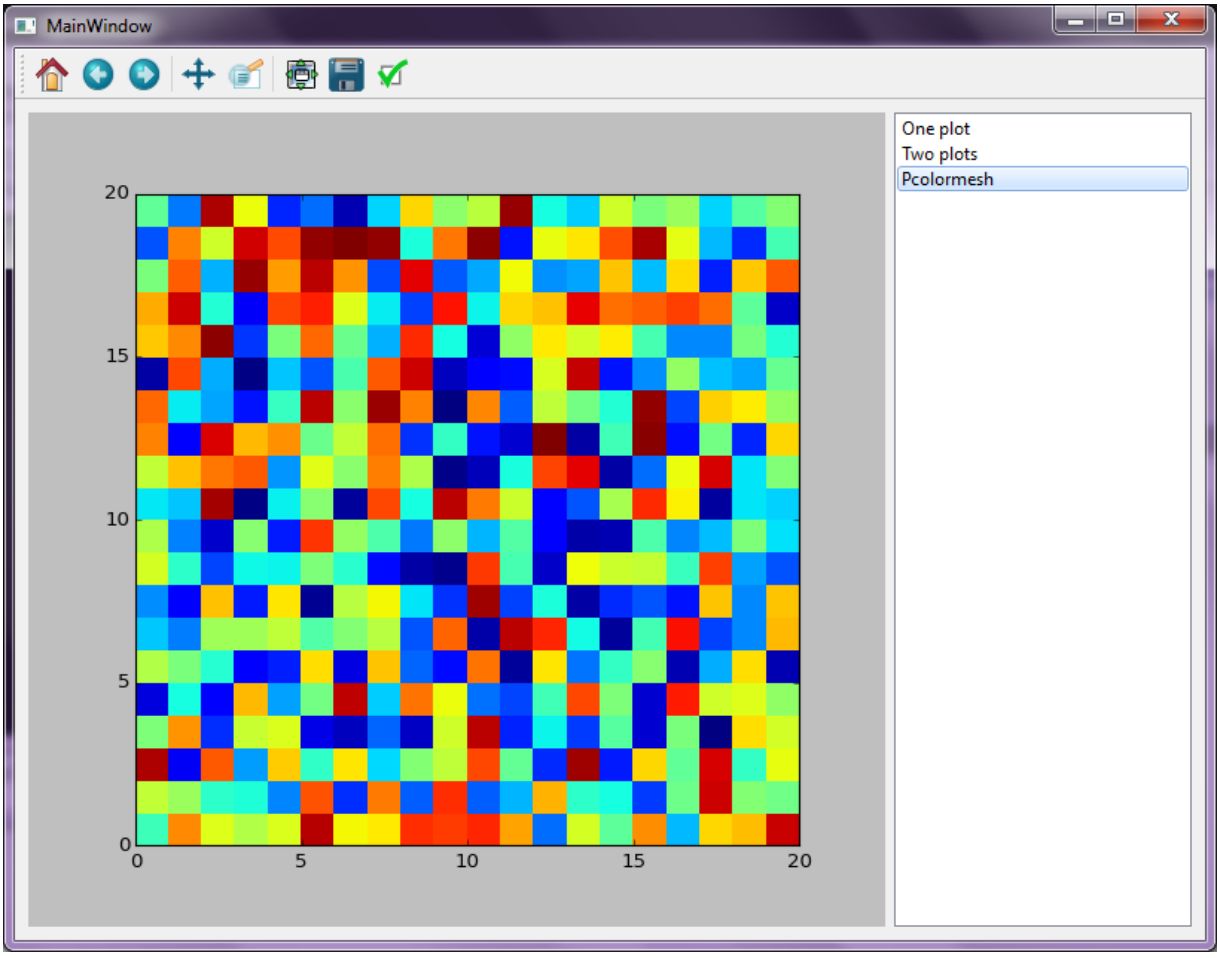
Notice that an empty figure instance was added to our plotting window in the initialization method. This is necessary because our first call to `changefig` will try to remove a previously displayed figure, which will throw an error if one is not displayed. The empty figure serves as a placeholder so the `changefig` method functions properly.

To test this, let's add a couple of Figure instances to our application to see what it does. We'll do this in the conditional code block at the end of the script.

```
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))

    fig3 = Figure()
    ax1f3 = fig3.add_subplot(111)
    ax1f3.pcolormesh(np.random.rand(20,20))

    app = QtGui.QApplication(sys.argv)
    main = Main()
    main.addfig('One plot', fig1)
    main.addfig('Two plots', fig2)
    main.addfig('Pcolormesh', fig3)
    main.show()
    sys.exit(app.exec_())
```

At first, when you run this code, you will be presented with a blank plotting window; however, select one of the plot names from the list widget and see what happens.



SUCCESS!!!

Running Our Application from IPython

IPython is an extremely popular and powerful tool for interactive data analysis. In addition, it is well designed to work with external GUI event loops, which makes it possible to run custom applications as well. To get this to work, we must tell IPython that we will be running an external Qt GUI using the `%gui qt` cell magic (version >2.4

of IPython). With IPython started, we can start our GUI application in the following manner.

Input:

```
1 %gui qt
```

Input:

```
1 import numpy as np
2 from matplotlib.figure import Figure
3 from custommpl import Main
```

Input:

```
1 main = Main()
2 main.show()
```

This will display our GUI with an empty plot window and figure list. We can recreate our example above by generating each figure and adding them to our application.

Input:

```
1 fig1 = Figure()
2 ax1f1 = fig1.add_subplot(111)
3 ax1f1.plot(np.random.rand(5))
4 main.addfig('One plot', fig1)
```

In this way, you can interactively process the data then add new figures to the GUI application as needed. This technique also works with IPython's qtconsole and locally-hosted notebooks.

February 20, 2015 Ryan Python IPython, Matplotlib, PyQt

2 thoughts on "Building a Matplotlib GUI with Qt Designer: Part 3"

Mark

May 13, 2015 at 8:55 am

This is by far the best tutorial on using matplotlib with qt-designer, thank you so much for taking the time to write it.

Nick

June 11, 2015 at 11:26 am

This has been very helpful for me creating a GUI using using matplotlib and Qt-Designer. I'm new to making GUI's and this tutorial was exactly what I needed to get started. Anyway you could post some information on plotting a real time graph in Qt-Designer?

Comments are closed.