Cute (Qt) Interfaces

For this class we'll be using the Qt (pronounced "cute") graphical interface framework. Qt is an open source, cross-platform C++ framework used (and supported) by organizations on the scale of LucasFilm, Siemens, and the US Army (we're all working for Uncle Sam now). This combination of open-source code and massive public support adds up to a project that will continue to improve and is likely be around for a long time.

As our language of choice is Python, we'll be using a module called PyQt, developed and maintained by Riverbank Computing. PyQt gives python bindings to all the compiled C++ functions of the full Qt framework. We'll be using the latest version, PyQt5 – which provides Python bindings to the latest version of Qt, version 5.

There's a good amount of documentation for PyQt5 online. Google searches will return a lot of helpful links. Just make sure you're looking at help for PyQt5 and not PyQt4 – or, if you can only find help for PyQt4, try to figure out where the modules you're looking for have been moved to in PyQt5.

Specifically, here are a few good links for learning and reference:

- ZetCode PyQt5 Tutorials
- PythonSpot PyQt5 Tutorials
- Riverbank's PyQt5 Reference Guide
  - Includes a list of differences between PyQt4 and PyQt5, useful for following PyQt4 tutorials
- Qt5 C++ Class Reference
  - Descriptions of all the classes and methods of the Qt framework

The Simplest Qt Application

Below is the most basic bit of code you need for a Qt application.

HelloQt-Functional.py

```
1  import sys
2  from PyQt5.QtWidgets import QApplication, QWidget
3
4  if __name__ == '__main__':
5      app = QApplication(sys.argv)
6      w = QWidget()
7      w.resize(250, 150)
8      w.move(300, 300)
9      w.setWindowTitle('Hello Qt')
10     w.show()
11     sys.exit(app.exec_())
```

Assuming you've got Python and PyQt5 installed correctly, this program should create an empty window with "Hello Qt" in the title bar. Believe it or not you've just made a complete windowed application. It doesn't do much at this point, but still, the Qt framework has invisibly taken care of a lot for you: connecting to your graphics hardware, to the operating system's multitasking scheduler, it's accepting user interaction (although you haven't configured your app to respond to any user actions just yet). If you alt-tab around your open applications you'll also notice yours, It should the be the one with no icon, or a placeholder icon, with the name "Python" or "Python 2.7".

Take a moment and go through this program to refresh yourself on the basic Python constructs. The important Qt-specific things to note here are the import statement on ***line 2***. Here you're digging into PyQt5 into the QtWidgets module – PyQt has a big heirarchy of modules and you'll need to learn where to find what, the class reference is useful for this. In general, you'll find many of the important classes in the modules `QtWidgets`, `QtCore` and `Qt`.

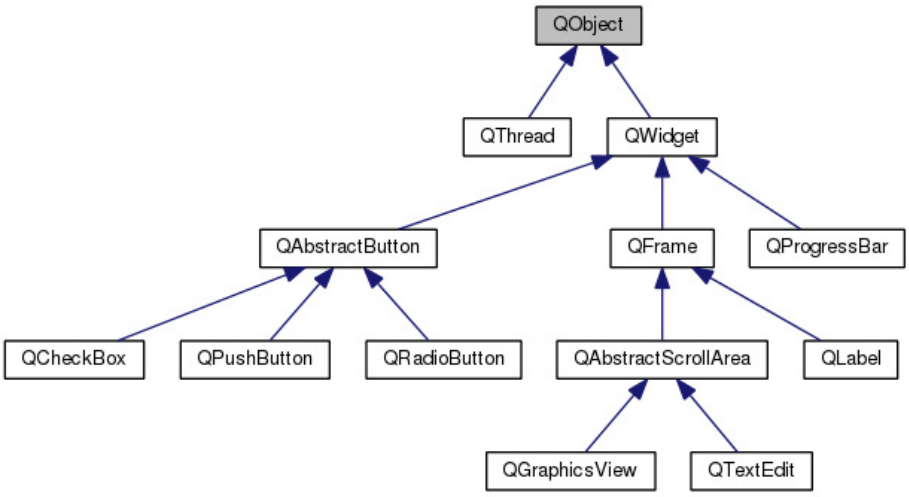
The code on ***line 5*** and ***11*** is also important. This is the boilerplate code for setting up a Qt application. In ***line 5*** you

create an instance of a `QApplication` (this is Qt's class that encapsulates all the code needed to get a GUI application up and running). Then on ***line 11*** you call the method `exec_()` on the application instance you created. This function starts Qt's event loop, at which point you take your hands off the interface you built and let it respond to user input.

Qt is Object Oriented

It's natural to think about the graphical user interface as being broken up into objects, their properties, and their behaviors. For example, a button (object) with a color and label (properties) and behavior (response to a click from the user). The whole object oriented paradigm (via SmallTalk) was developed by Alan Kay & his colleagues at Xerox PARC specifically to provide a suitable language for talking about GUIs.

It should be no surprise, then, that Qt is deeply object oriented. Consider this diagram of the object heriarchy in Qt, showing `QWidget` as the parent class of many graphical user interface elements, like buttons, scrollboxes, text fields, even graphics canvases.



The creators of Qt intended people who use the framework to think in a certain way, and that way is object-oriented. In Qt the class `QWidget` is the parent class of all things graphical, it can be everything from a window to a button to a 3D drawing canvas. The standard way of creating a Qt app is to begin by extending `QWidget` or one of its subclasses, and using that as your application window. Here's an example rewriting our first application by extending `QWidget`.

HelloQt-oop.py

```
1  import sys
2  from PyQt5.QtWidgets import QApplication, QWidget, QLabel, QPushButton
3
4  class MyMainWindow(QWidget):
5      def __init__(self):
6          super(MyMainWindow, self).__init__()
7          self.initUI()
8
9      def initUI(self):
10         self.resize(250, 150)
11         self.move(300, 300)
12         self.setWindowTitle('Hello Qt')
13         self.show()
14
15 if __name__ == '__main__':
16     app = QApplication(sys.argv)
17     win = MyMainWindow()
18     sys.exit(app.exec_())
```

The basic object-oriented syntax here should be familiar to you. Make sure you know how to define a class, what an object and an instance are and how you create new instances, what methods and properties of a class are, and what the `__init__` method does. If you don't know this then go spend some time reviewing object oriented programming in python, because you'll need to know this stuff to use PyQt!

There isn't anything too shocking in this program once you understand the previous one. Instead of creating an instance of `QWidget` I've instead made my own custom subclass of `QWidget`, called `MyMainWindow`. After running all the initialization code of `QWidget`, my subclass runs its own initialization code (the `initUI` method) that sets up the window to look the way I want. In this case, I just took the calls to `resize move setWindowTitle` and `show` and moved them inside `initUI`, so that the window instance automatically calls these methods on itself when it's created.

The main program is now reduced to the two lines of Qt boilerplate code (***16 & 18***) with one line in between that creates my custom application window. All the real meat of the program is shifted into the classes, which represent entities within the user interface. We haven't gotten to interaction yet, but when we do you'll see that each user interface object will also include all the code necessary to react to the different actions a user can perform on it. This quality of being able to 'black box' the behaviors of a widget into its class is called 'encapsulation', one of the main conceptual benefits of using OOP to build GUI apps.

Windows & Widgets

Here's a more interesting example with multiple `QWidget` subclasses. I'm using two png images `safariextz.png` and `safariextz_inv.png` which need to be in the same directory as your python program for this to work. You can find them in the course github repo in the folder `01-Intro`.

HelloWidgets.py

```
1  import sys
2  from PyQt5.QtWidgets import QApplication, QWidget, QLineEdit, QPushButton, QLabel
3  from PyQt5.QtGui import QPixmap
4
5  class MyWindow(QWidget):
6      def __init__(self, color="white", xpos=0, ypos=0):
7          super(MyWindow, self).__init__()
8          self.initGUI(color, xpos, ypos)
9
10     def initGUI(self, color, xpos, ypos):
11         self.setGeometry(0, 0, 260, 260)
12         self.setWindowTitle("Safari Extension")
13         self.button1 = QPushButton(self)
14         self.button1.setText("Help!")
15         self.text1 = QLineEdit(self)
16         self.text1.setText("Hello Widgets")
17         if(color == "white"):
18             pix1 = QPixmap("safariextz.png")
19         else:
20             pix1 = QPixmap("safariextz_inv.png")
21         self.img1 = QLabel(self)
22         self.img1.setPixmap(pix1)
23         self.text1.move(20, 200)
24         self.button1.move(170, 180)
25         self.move(xpos, ypos)
26         self.show()
27
28 if __name__ == "__main__":
29     app = QApplication(sys.argv)
30     mywin1 = MyWindow("white")
31     mywin2 = MyWindow("inv", 360, 0)
32     mywin3 = MyWindow("white", 0, 350)
33     mywin4 = MyWindow("inv", 360, 350)
34     sys.exit(app.exec_())
```

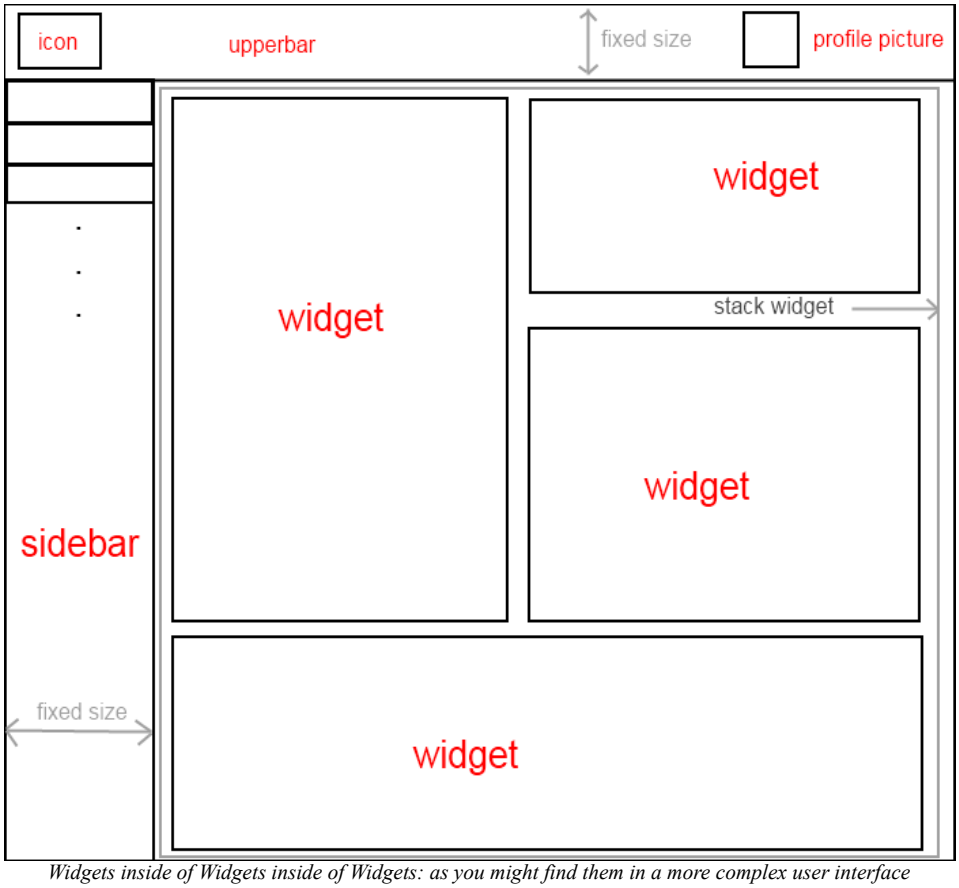
Once you've got this up and running you should see four windows open in a grid, each with an image (`QLabel`), a button (`QPushButton`), and a text field (`QLineEdit`) inside of it. That's four instances of the custom `QWidget`

class `MyWindow`, each moved to a unique location on the screen based on the input arguments to the `__init__` method when the instance is created.

Besides using a few new user interface classes, this program illustrates another important aspect of working with widgets: they can be nested inside of each other. If you think of a graphical user interface, all your widgets, like buttons and drop-down menus, exist inside of a window. The window itself is also a widget, it's the widget at the very top of the heirarchy that encloses all the other widgets, like a canvas. You can also have complex widgets built up from simpler widgets, and embed that into a window. In general, the enclosing widget is called the "parent" and the inner widget is called a "child", it's the same terminology used with classes, but here it means that the child widget is positioned inside the coordinate system of the parent widget.

In general, to nest one widget inside of another, you need to pass the parent widget in as an argument to the `__init__` method of the child widget when you create it. You can see examples of this in the program above, in ***lines 13 15 and 21.*** In those three examples, the parent widget is the instance of `MyWindow` (`self`) and the button, text field, and label are all child widgets inside of the window.

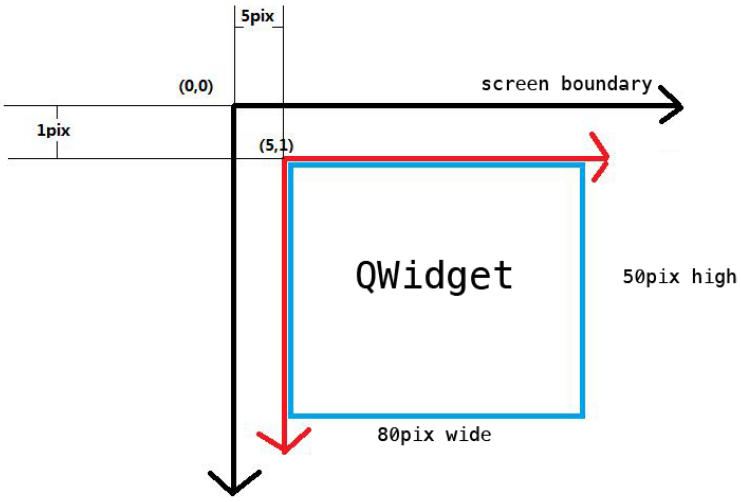
If you don't specify a parent widget when creating a new `QtWidget` instance, Qt will assume that your new widget will be a stand-alone window. Notice that we don't make an argument for a parent widget in the `__init__` method of `MyWindow`. This is because I'm intending it to be used as a window.



Widgets inside of Widgets inside of Widgets: as you might find them in a more complex user interface

Notice in ***line 23 & 24*** we're setting the x, y position of the text field and the button. These coordinates are *inside* the coordinate system of the enclosing window. What this means is that 0,0 is the upper left corner of the window, not the screen.

Note: *Your screen is made up of some N x M resolution in pixels. Qt calculates screen positions starting from the top left corner of your screen (coordinate 0,0) and counts rightward in the X direction and downward in the Y direction.*



The result of calling `setGeometry(5, 1, 80, 50)` *on a* `QWidget` *instance*

What Next?

Try to make an interface by arranging basic UI elements from the Qt framework. Look around the class reference to see what kinds of subclasses of `QWidget` there are. We haven't looked at responding to user input yet, so focus on getting a feel for how to create and arrange GUI elements within a window. Try to think aesthetically, and use this as an opportunity to review the basics of Python, things like for loops, classes, and functions.