Writing out Tkinter program as a class.

In previous lectures, we wrote a small Tkinter function, which draws a rectangle wherever a user clicks on a canvas.

```
import Tkinter as Tk #or tkinter in v.3
root = Tk.Tk()
canvas = Tk.Canvas(root, width=500, height=500)
canvas.pack()
def rectangle(ev):
    canvas.create_rectangle(ev.x,ev.y, ev.x+20, ev.y+20, fill="blue")
canvas.bind("<Button-1>", rectangle)
root.mainloop()
```

Writing these types of programs as classes has a lot of advantages. For example, it makes it easy to instantiate the class with different parameters, and keep track of local variables as object attributes. As a class, this program might look something like this:

```
import Tkinter as Tk #or tkinter in v.3

class RectangleGUI:
    def __init__(self, master):
        self.master = master
        self.canvas = Tk.Canvas(root, width=500, height=500)
        self.canvas.pack()
        self.canvas.bind("<Button-1>", self.rectangle)

    def rectangle(self,ev):
        self.canvas.create_rectangle(ev.x, ev.y, ev.x+20, ev.y+20, fill="blue")

root = Tk.Tk()
gui=RectangleGUI(root)
root.mainloop()
```

As before, the `root` is the main window of our application. The `root.mainloop()` command let's the main window run until the user quits it. Widgets always have parent (master) widgets, all the way up to the root. In this case, we have only one widget, the canvas, which has the root as its parent.

Using Qt Designer

Qt Designer is a general (not specific to python) software for designing user interfaces. It lets you design the interface by dragging and dropping widgets, and editing their properties in a environment that is easy to oversee. In Qt Designer, we only control the design part, though. We have to return back to python in order to make the interface actually do anything.
Qt Designer is generally very easy to use, and most of the options are self-explanatory. You should play around and figure out what is available. A few things to keep in mind: The vertical and horizontal lay-outs are extremely useful. They keep the widgets in a nice pattern, even if the user resizes the window. Also, they make it easy for you to add custom widgets in your code later on. You can simply add them to a lay-out and they will automatically be arranged with respect to the other widgets. It is very important to name your widgets here in Qt Designer, since this is where you can see which one is which (especially once you create many of the same types of widgets). Once this has been translated to python, it is hard to figure out which one is which.

Once you have created your design, you can save it as a `design.ui` file. (You should give it a more descriptive name, of course.) We then use PyQt from the command line (not from Spyder) to transform this into a python file. Remember that Qt Designer is not python specific, but the PyQt package translates files from one to the other. Make sure that the path in your command line matches where the file is saved, and run the line:

```
pyuic design.ui -o design.py
```

The `design.py` contains python code that creates a class, which defines the design. We copy the code from this file to a new file `main.py` (again giving it a more descriptive name) which will be the file that actually does something. The reason to copy is that if we change anything about the design in Qt Designer and rerun the command line command, then the `design.py` will be overwritten, and lose any code that we added to it (code that makes it do stuff). We could of course also import the code from `design.py` into `main.py`, but we do want to edit some of the code that was created in the former in the future. This will make sense once we add custom widgets for which there is no option in Qt Designer. The basic code for the "method" part of our `main.py` looks as follows:

```
import sys
class MyGUI(QtGui.QMainWindow, Ui_MainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        Ui_MainWindow.__init__(self)
        self.setupUi(self)

if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    window = MyGUI()
    window.show()
    sys.exit(app.exec_())
```

Notice that much of the MyGUI class is directly taked from `QtGui.QMainWindow` and `Ui_MainWindow`, which were both defined in the design part. The `setupUi` attribute comes from there. In the next part, the line `if __name__ == "__main__":` looks a little strange, but this runs the code directly from the user double clicking on the file. This means that a user can use your GUI without seeing any python code. To do this, we need to interact directly with the operating system, which is where the `sys` module comes in.

A simple GUI

We start by creating a very simple GUI, which lets the user enter two numbers, press a button, and then see the sum of the two numbers. To do this, we create a design in Qt Designer, which has three line edit widgets ("first", "second", and "result"), and one push button ("plusbutton"). It is useful to keep the documentation handy, so you can see which attributes you can use for the different types of widgets. The method part of the code now looks as follows:

```
import sys

class Plus(QtGui.QMainWindow, Ui_MainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        Ui_MainWindow.__init__(self)
        self.setupUi(self)
        self.plusbutton.clicked.connect(self.addit)

    def addit(self):
   a=float(self.first.text())
        b=float(self.second.text())
        s=str(a+b)
        self.result.setText(s)

if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    window = Plus()
    window.show()
    sys.exit(app.exec_())
```

Go through the code and make sure you understand how the object attributes of the different widgets are used. You need to create the design part yourself.

A simple plotting GUI

Now we will create a GUI that uses a custom widget. This widget is a canvas and figure from the `matplotlib` module, which has built-in functionality for GUIs, but these are not included in Qt Designer (they are too specific to this python library). We create a code that lets the user pick a color and a parameter, b, and then plots the function $\sin(bx)$ in the chosen color. The full code is given here, but these notes will not go over every line in detail like we did in the lecture. You should spend some time playing around with this, and adapting it. Notice the highlighted lines in the design part of our code: this is the part we edited by hand!

```
from PyQt4 import QtCore, QtGui
from matplotlib.figure import Figure
from matplotlib.backends.backend_qt4agg import (
FigureCanvasQTAgg as FigureCanvas,
NavigationToolbar2QT as NavigationToolbar)
import sys

try:
    _fromUtf8 = QtCore.QString.fromUtf8
except AttributeError:
    def _fromUtf8(s):
        return s
try:
    _encoding = QtGui.QApplication.UnicodeUTF8
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig, _encoding)
except AttributeError:
    def _translate(context, text, disambig):
        return QtGui.QApplication.translate(context, text, disambig)

class Ui_MainWindow(object):
    def setupUi(self, main):
        main.setObjectName(_fromUtf8("main"))
        main.resize(800, 600)
        self.centralwidget = QtGui.QWidget(main)
        self.centralwidget.setObjectName(_fromUtf8("centralwidget"))
        self.vertical_layout = QtGui.QVBoxLayout(self.centralwidget)
        self.vertical_layout.setObjectName(_fromUtf8("vertical_layout"))
        self.green = QtGui.QRadioButton(self.centralwidget)
        self.green.setChecked(True)
        self.green.setObjectName(_fromUtf8("green"))
        self.buttonGroup = QtGui.QButtonGroup(main)
        self.buttonGroup.setObjectName(_fromUtf8("buttonGroup"))
        self.buttonGroup.addButton(self.green)
        self.vertical_layout.addWidget(self.green)
        self.red = QtGui.QRadioButton(self.centralwidget)
        self.red.setChecked(False)
        self.red.setObjectName(_fromUtf8("red"))
        self.buttonGroup.addButton(self.red)
        self.vertical_layout.addWidget(self.red)
        self.lineEdit = QtGui.QLineEdit(self.centralwidget)
        self.lineEdit.setObjectName(_fromUtf8("lineEdit"))
        self.vertical_layout.addWidget(self.lineEdit)
        self.gobutton = QtGui.QPushButton(self.centralwidget)
        self.gobutton.setObjectName(_fromUtf8("gobutton"))
        self.vertical_layout.addWidget(self.gobutton)
        self.figure = plt.figure()
        self.canvas = FigureCanvas(self.figure)
        self.vertical_layout.addWidget(self.canvas)
        main.setCentralWidget(self.centralwidget)
        self.menubar = QtGui.QMenuBar(main)
        self.menubar.setGeometry(QtCore.QRect(0, 0, 800, 20))
        self.menubar.setObjectName(_fromUtf8("menubar"))
        main.setMenuBar(self.menubar)
        self.statusbar = QtGui.QStatusBar(main)
        self.statusbar.setObjectName(_fromUtf8("statusbar"))
        main.setStatusBar(self.statusbar)

        self.retranslateUi(main)
        QtCore.QMetaObject.connectSlotsByName(main)

    def retranslateUi(self, main):
        main.setWindowTitle(_translate("main", "Root Plot", None))
        self.gobutton.setText(_translate("main", "Go!", None))

#############################################
```

```
import sys
class SinGui(QtGui.QMainWindow, Ui_MainWindow):
    def __init__(self):
        QtGui.QMainWindow.__init__(self)
        Ui_MainWindow.__init__(self)
        self.setupUi(self)
        self.gobutton.clicked.connect(self.plotfnct)

    def plotfnct(self):
        b=float(self.lineEdit.text())
        fig = Figure()
        ax = self.figure.add_subplot(111)
        x=np.arange(-np.pi,np.pi,.001)
        y=np.sin(b*x)
        ax.hold(False)
        if self.green.isChecked():
            ax.plot(x,y,color='g')
        else:
            ax.plot(x,y,color='r')
        self.canvas.draw()

if __name__ == "__main__":
    app = QtGui.QApplication(sys.argv)
    window = SinGUI()
    window.show()
    sys.exit(app.exec_())
```