So Sing Us a Song…

For this lab, you're going to turn your computer into a piano! Along the way, we'll learn a bit about writing a Graphical User Interface (GUI) and put classes to work for us.

This lab will also get you going on the homework assignment. It looks long, but that's because I'm giving you most of the code here.

Since I can't access my website on the road, I've posted the starter code for this assignment in Blackboard. Go download the ZIP file and extract it into the directory where you are going to work. Never extracted a ZIP file? It's not a big deal, just open it up and drag all the files out into a normal directory.

ZIP is a compressed archive – it writes the contents of many files all into one file and *compresses* it. That is, it alters the *encoding* of the data so that it takes up less space.

Part I: Gooey

We've already open windows and drawn images in them. You make menus and buttons and drop-down boxes the same way, by showing images of them in the window. However, images are the wrong level of abstraction here. We want to think about text labels and buttons and such instead.

In GUI libraries, these things are often called *widgets*. The built-in Python Tkinter library has everything you need, but it also uses a few language features we haven't talked about yet. To avoid that, I've made a simple wrapper class for you to use. To see a window, start a new file (in the same directory with the starter files, of course) with this:

```
from simplewin import simplewin

# create a new window object
win = simplewin()

# and give it control
win.run()
```

Make sure that works! It should open an empty window, and that means you're set up right.

STOP AND RUN!

The *simplewin* class has an *add_label* method that takes a string and adds that string as a label (static text) to the window. Try add a few labels before calling *win.run()*.

STOP AND RUN!

Labels don't do anything, so let's work with a button instead. The button widget in Tkinter has a lot of built-in functionality, including making it look like you pressed it when you click on it.

GUI code is *event-driven*. That is, it doesn't just run top to bottom like the programs we've written so far, it has to respond to user actions, like pressing a button. The way that Tkinter (and most GUI libraries) do this is with *callback functions*.

simplewin has an *add_button* method that also takes a string like *add_label*, but the second argument is something new. The second argument is the name of a function. When the user clicks on the button, that function will be called. Passing a function as a parameter to be called later is a *callback*.

For example, try adding this code to you project. (Put the new function up at the top to be neat, and put the call to *add_button* right before the call to *win.run()*):

```
def example_callback():
      print("I have been called…back")

win.add_button("Test Button", example_callback)
```

Run it and click the button, and you should see the output printed to the console.

STOP AND RUN!

Pay close attention! Notice that when passing *example_callback* into the *add_button* method, I did not put () after it. That's because putting () at the end of a function name tells Python to call that function right then and there. I don't want to do that! I want to pass it in to be called later.

Your turn. In order to quit a GUI program, you need to call *win.quit()*. Write a function that prints "Goodbye!" to the console and then calls *win.quit()*. Add another button to the window that calls back that function you just wrote.

When you click the new button, you should see "Goodbye!" in the console, and the window should close.

STOP AND RUN!

Clicking buttons is all well and good, but what about pressing keys?! In the same way, we can set up a callback function whenever the user presses a key (or clicks the mouse, although we won't worry about that here).

simplewin has a method called *add_key_handler* that takes a function to call when the user presses a key. Start with this code:

```
def key_down(event):
      print("Warning! A key has been pressed!")

win.add_key_handler(key_down)
```

(Again, set the handler before the *win.run()* call. *win.run()* indicates that you are done and he window can take over looping forever waiting for events and making callbacks.) Run it and push some keys!

STOP AND RUN!

Somewhat useful, but more so if you knew what key had been pressed… Notice that the key_down callback takes an argument, unlike the button press callbacks we did before. Callback functions must match the parameters of the callback situation. Button presses in Tkinter pass no parameters. Key presses pass a single parameter, which is an object with information about the key press.

In particular, we are interested in which key was pressed. Change that print statement to the following and run it again.

```
      print("Warning! {} has been pressed!".format(event.keysym))
```

Now run and push keys, and you've got all the GUI stuff you need.

STOP AND RUN!

Part II: Sing Us a Song Tonight…

Python has a very simple built-in sound library for windows called *winsound*. (Mac laptops, see below.) In the starter code, I provided a bunch of *wav* sound files which are piano notes. In order to play one, you just do:

```
import winsound

winsound.PlaySound("C.wav", winsound.SND_FILENAME | winsound.SND_ASYNC)
```

Try it (before you call *win.run()*) to make sure it works. Make sure your volume is turned up. The extra arguments there are called a *bitmask* or *flags*, and they are used to tell *PlaySound* that you're specifying a sound file by name, and not to wait until the sound finishes before moving on. Nothing to worry about right now.

IF YOU ARE ON A MAC, try this instead. I can't test it on a Mac, unfortunately. For lab today, if it fails, switch to working on a lab computer.

At a terminal command prompt: *py -3 -m pip install simpleaudio*

```
import simpleaudio as sa
wave_obj = sa.WaveObject.from_wave_file("C.wav")
wave_obj.play()
```

STOP AND RUN!

Okay, so a little less hand holding here. You know how to play a sound, and you've already written a callback function that can tell what key the user pressed.

Update your callback function so that the a-s-d keys on your keyboard play the C-D-E note files. Then play that timeless classic, hot cross buns!

STOP AND RUN!

Part III: Ring the Bell! Class (sp) is In!

Yeah fine, it's "school". Whatever. This is about classes.

Start a new Python file for this part.

You could solve that last part of part II with a simple if/else tree. Or you may have gone farther than that already. The point is, it's not a great solution when you start thinking about doing all 13 keys in an octave (C to the next C). What we really want to do is make this *data-driven*; keep lists of keys and notes rather than repeat the same if/else code over and over.

This is also a place where classes can make our code a lot neater.

Update 1: The Piano

Classes are used to represent objects, with data and functionality. One of the objects in our problem here is a piano, with a bunch of keys. Life would be easier if we could just say, "Hey piano, play key 4" or key 7, or whatever. Let's make a class to do that.

Declare a *piano* class with one data member, a list of the notes (the wav filenames) in this piano, starting with C and ending with C2. Note that, for example, "Cs.wav" is C#. Then write a method called *play* that takes one argument, the index of the note you want to play (start with 0 of course). Use that index to look up the filename

for *PlaySound*. (hint: you should not have a big if/else statement in your *play* method.) You should be able to test your piano by doing:

```
p = piano()
p.play(6)
```

STOP AND RUN!

We've talked about putting variables and code into a *main* function rather than having them sitting out at the global level. Another option is to create a class that represents the whole program. In this part, you're going to replace the global code with this *app* class and finish it. Notice that the window setup code that was at the global level is now in a *run()* method, which is called to run the program. Notice also that win is now a data member of the class rather than a global variable, and the callback functions *key_down* and *quit_program* are now the methods *self.key_down* and *self.quit_program*. Everything is contained within this class.

Copy this class into your new file with your *piano* class. Transfer the code from your previous *key_down* function into the *key_down* method and it should still work.

```
class app:
    def __init__(self):
        # the special keyword "None" indicates no object
        self.win = None

    def run(self):
        self.win = simplewin()

        self.win.add_label("My Piano")
        self.win.add_button("Quit", self.quit_program)

        self.win.add_key_handler(self.key_down)

        self.win.run()

    def key_down(self, event):
        # your key_down code here

    def quit_program(self):
        print("Goodbye!")
        self.win.quit()

app = app()
app.run()
```

STOP AND RUN!

Last part, let's finish the whole octave without a big if/else statement in *key_down*.

The *piano* class you made can play notes based on index. That's good, because computers are great with numbers. We need to make our *app* class work the same way.

In the *app* constructor, add two more data members. A *piano* object, and a list of the keyboard key names that map to the piano keys. Both white and black keys, so it should include a-w-s-e-d-f-t-g-y-h-u-j-k. In the *key_down* method, we know what key the user pressed, by name. Update that method to:

1) Look up the index of the key that was pressed in the keys list in *app*
2) If it was a valid key (in the list), then have the *piano* object play that note (by index)

By maintaining the list of keys in *app* and the list of notes in *piano*, we make our program data-driven. That is, those lists of data determine the behavior of the program (which key maps to which note), rather than having it fixed in code (the big if/else tree).

You should now be able to play your full octave (white and black keys)! Perform a song for your neighbors, then submit both code files for this lab. Hang onto this, because you're going to extend it for the homework.