CS 1110, LAB 4: ASSIGNMENT 1

http://www.cs.cornell.edu/courses/cs1110/2014fa/labs/lab04.pdf

First Name: ____________ **Last Name**: ____________ **NetID**: ________

Today's lab is an open office hour to work on Assignment 1. Take advantage of it to get whatever last minute help that you might need. If you have finished Assignment 1, we have an optional exercise for you to work on (see below). However, that is not required.

Since many students often wait to the last minute, the consultants will be fairly busy during this lab. If you feel that you are being ignored (e.g. you have held your hand up for several minutes with no one recognizing you), approach one of the course staff and ask about "getting a place in line".

Getting Credit for the Lab. Because you are working on the assignment, you will receive full credit for this lab if you turn in the assignment on time (e.g. Thursday before midnight). There is nothing else to show to you instructor. You do not even need to swipe your card this time.

If you wish, you can show the optional exercise to the instructor. However, that is not required.

1. Lab Exercises (OPTIONAL)

All the exercises this week are completely optional. They are designed to give you practice with objects, the new concept introduced this past week. If you choose to work on the lab, there are three files to download from the Labs section of the course web page.

- `window.py` (a module that provides access to Window objects)
- `tuple3d.py` (a module that provides access to Point objects)
- `lab04.py` (a module with several functions on Points)

Once again you should create a *new* directory on your hard drive and download all of the files into that directory. Alternatively, you can get all of the files bundled in a single ZIP file called `lab04.zip` from the Labs section of the course web page.

2. Windows (OPTIONAL)

As we said in class, a type is a collection of *values* together with its operations. Normally when we think of values, we think of things like numbers, or `True` and `False`. But these values could take on any form. A value could be a window on your screen. That is the motivation for the type `Window`. In this part of the lab, you will create and manipulate `Window` objects.

All of the types that we have seen so far have natural ways to represent their values. We represent ints by whole numbers; we represent strings as characters inside double quotes. But there is no natural way to represent values of type `Window`. For this type, we have to use a special functon – called a constructor – to create a new object of that type.

Course authors: D. Gries, L. Lee, S. Marschner, W. White

The name of a constructor function is generally the same name as the type. The type `Window` is provided by the module `window` (which you should have downloaded). Import this module and enter the assignment statement

```
w = window.Window()
```

What happens? What is stored in the varible `w`?

2.1. **Window Attributes.** As we discussed in class, attributes are named variables that are stored inside an object. You can use attributes in expressions, or even assignment statements.

One of the interesting thing about GUI objects is that assigning new values to an attribute can have visible effects. In the table below we have a list of expressions and assignment statements. **Enter these into the Python shell in exactly the order presented**. If it is an expression, give (or guess) the value that Python returns. If it is an assignment statement explain (or guess) the result of the assignment.

Statement or Expression	Expected Result	Actual Result	Reason for Actual Result
`w.x`			
`w.x = 100`			
`w.y`			
`w.y = 100`			
`w.width = 10`			
`w.height`			
`w.title`			
`w.title = 'window'`			

2.2. **Window Methods.** Window objects also have methods. Unlike string methods, which are functions, these methods are procedures that do something to the object. Execute calls for the three methods shown in the table below. Explain what happens when you call them.

Method	**Result When Called**
`w.beep()`	
`w.iconify()`	
`w.deiconify()`	

2.3. **Positioning a Window.** You have already seen the Window size and position is controlled by attributes. For the Window whose name is in `w`, look at the attributes for the x-coordinate and y-coordinate. Write their values here:

Next, create a second Window object, storing its name in another variable. This should pop up a new window. What are the x-coordinate and y-coordinate for this window?

Is there something unusual about how screen coordinates work? What do you notice about the difference in coordinates between the two windows?

2.4. **Resizing a Window.** Create a new Window, storing its name in variable `w`. Try resizing the Window with your mouse to make it bigger. Look at the attribute function `resizable` of `w` to see whether the Window is resizable. What is the value of this attribute?

Now execute the assignment

```
w.resizable = False
```

Try resizing the Window whose name is in `w` with your mouse. Is it resizeable now?

Assign the attribute `resizable` to True. Once you have done that, call the procedure

```
w.setMaxSize(50,100)
```

What happened to your Window?

What happens when you try to resize this Window?

3. Test the Procedure `cycle_left(p)` (OPTIONAL)

This part of the lab is having you write another unit test, just like you did for Lab 3, and are currently doing for Assignment 1.

Instead of strings, this time you will work with a Point object. We introduced the type Point in lecture; objects of type Point are points in 3-dimensional space. They have three attributes, `x`, `y`, and `z`, corresponding to the three spatial coordinates, stored as floats. You create Point objects with a constructor call, supplying three arguments to set the coordinates `x`, `y`, and `z`. For example, the constructor call

```
Point(2,1,0)
```

creates a Point object with (x,y,z) = (2.0, 1.0, 0.0) and returns the id of the object (what is written on the folder tab on the left).

To use the class Point, you must `import` the module `tuple3d`. Therefore you have to preface the constructor call above with the prefix `tuple3d`, as you would for any function in the module.

The function `cycle_left(p)` is actually a procedure. It does not return anything. Instead, this procedure changes the contents of the object (e.g. the folder) whose name is in `p`. Read the specifications of this procedure to understand what it does.

3.1. **Create a Unit Test Script.**

We have not provided you with a test script file this time. You should create your own test script this time and call it `test_lab04.py`. It should have the same format as the other test scripts that you have worked with. Remember to import both `cornelltest` and `tuple3d`, as you will be testing points.

Once your unit test file is working, add the test procedure `test_cycle_left()`. Once again, this test procedure should start out with a simple print statement so that you can verify that it is working properly. You should also add a call to this test procedure in the script code, before the final print statement.

3.2. **Implement the First Test Case.**

This procedure should take a point, and "shift" all of the coordinates to the left (with the x coordinate moving to the z coordinate). To test this out, you need to add the following code to `test_cycle_left()`.

- Create a `Point` object (0,0,1) and save its name in a variable `p`.
- Call the procedure `cycle_left(p)`.
- Test that `p` is now the point (0,1,0).

The last step requires further details. You cannot write

```
p == (0,1,0)
```

This will return `False`. That is because (0,1,0) is not a Point object. It is a value of a type that we have not yet seen in class (and will not see for a while). Instead, you have to check each of the attributes — x, y, and z — separately.

Remember that the attributes of `p` are all floats. Therefore, you want to use the function `assert_floats_equal()` to check that the values are all correct. So, to check that `p` is the point (0,1,0), you would add the following statements:

```
cornelltest.assert_floats_equal(0,p.x)
cornelltest.assert_floats_equal(1,p.y)
cornelltest.assert_floats_equal(0,p.z)
```

Add these test cases to the test procedure `test_cycle_left()` and run the unit test script. There should not be an error this time; check your test procedure if you run into any problems.

3.3. **Add More Test Cases for a Complete Test.** Obviously, the point (0,1,0) is not enough to test this function. We told you there was an error, and you have not found an error yet. Why is this point not sufficient to test the function shift?

What are good points for testing out this function?

Implement the test case(s) above, and run the script again. You should get an error message now.

3.4. **Isolate the Error.** Unit tests are great at finding whether or not an error exists. But they do not necessarily tell you where the error occurred. The procedure `cycle_left()` has three lines of code. The error could have occurred at any one of them.

In the optional part of Lab 3, we saw how to use `print` statements to help us isolate an error. We use them to *inspect* a variable immediately after we have assigned a value to it.

Open up `lab03.py`. Inside of `cycle_left`, after the assignment to `p.x`, add the statement

```
print p.x
```

Do the same after the remaining two assignments (that is, print `p.y` and `p.z`). Now run the script. Before you see the error message, you should see three numbers print out. Those are the result of your print statements. These numbers help you "visualize" what is going on in `cycle_left()`.

There should be enough information that you can tell which value printed out is the one assigned to `p.y`. How do you tell this?

3.5. **Fix and Test.** You should now have enough information from these three print statements to see what the error is. What is it?

Fix the error and test the procedure again by running the unit test script.

3.6. **Clean up `cycle_left()`.** As in Lab 3, you should never leave print statements in your code once you are finished with debugging. Remove all of the print statements added for debugging. You can either comment them out (fine in small doses, as long as it does not make your code unreadable), or you can delete them entirely.

Run the unit test script one last time, and you are done.