CS 1110, LAB 4: ASSIGNMENT 1

http://www.cs.cornell.edu/courses/cs1110/2016fa/labs/lab04/

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. From the past three labs, you should have most of what you need to work on this assignment.

The only thing new in Assigment 1 is the test script. You are welcome to try to figure this out on your own. However, if you are struggling with the test script, we have a few activities in this lab to help you along. **All of these activities are optional** and are not required to get credit on the lab. We are just trying to make sure that you get as much help as possible.

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. Sunday before midnight). There is nothing else to show to you instructor. You do not even need to swipe your card this time.

There is no online component for this lab. The web service will indicate that you completed lab if you submit the assignment on time. If you do decide to complete the optional exercise, you may show that to your instructor to check that it is correct.

Lab Materials (OPTIONAL)

If you do want to work on the optional exercises, there are some files for you download. You can get all of these files from the Labs section of the course web page.

http://www.cs.cornell.edu/courses/cs1110/2016fa/labs

For today's lab you will notice two files.

- `funcs.py` (a module with some questionable functions)
- `lab04.py` (a testing script)

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. On both Windows and OS X, you can turn a ZIP file into a folder by double clicking on it. However, Windows has a weird way of dealing with ZIP files, so Windows users will need to drag the folder contents to *another* folder before using them.

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

1. Working with a Test Script (OPTIONAL)

In the previous lab, you tested your function by typing a few examples into Python using the interactive mode. This works if you only have one or two simple functions. For more complex software, you need to learn how to automate the process using a *script*.

As we saw in the second lab, a script looks like a Python module, except that we do not import scripts. We run them directly from the command line. The file `lab04.py` is a script. To run this file, **navigate the command line to the folder with this file**, but do not start Python (yet). When you are in the right folder, type the following:

```
python lab04.py
```

What did the script print to the screen when you ran it?

Now open up `lab04.py` in Komodo Edit. As with the test script in class you will notice two things: a **test procedure** and the **script code**. A test procedure is a function that uses the `cornelltest` module to test *other* functions. The script code contains a call to this test procedure, as the procedure cannot do any testing if you do not call it.

Actually, the test procedure `test_asserts` does not actually test another function. It is just a random collection of assert functions to show you what you can do with the `cornelltest` module. In particular, you will see the three functions `assert_equals`, `assert_true`, and `assert_floats_equal`.

For right now, we are going to focus on `assert_equals`, which is the most important of the three. This function compares the answer that you expect (a value) with the answer that you compute (an expression) and makes sure that they are the same. If they are the same then *nothing happens*. Otherwise, the function will quit Python and inform you that there is a problem.

Let us see what happens when something unexpected is received. Inside of the test procedure `test_asserts`, uncomment the line

```
cornelltest.assert_equals('b c', 'ab cd'[1:3])
```

Run `lab04.py` as a script again. You will see answers to *three important debugging questions*:

- What was (supposedly) expected?
- What was received?
- Which line caused `cornelltest.assert_equals` to fail?

What are the answers to three questions above?

Add the comment back to that line so that it is no longer executed (and so there is no error). Then uncomment the line at the end of the test procedure:

```
cornelltest.assert_equals(6.3, 3.1+3.2)
```

Run the script one last time and look it what happens. Based on the result, explain when you should use `cornelltest.assert_floats_equal` instead of `cornelltest.assert_equals`:

Recomment this last line of the test procedure before going on to the next activity.

2. The Function `has_a_vowel(s)` (OPTIONAL)

Now that you know how test scripts work, it is time to create a unit test script to check for any errors in the module `lab03`. You are going to start by testing the function `has_a_vowel(s)`. We guarantee that this function has a bug in it.

2.1. **Creating a Test Procedure.** Following the naming convention showed in class, you should test `has_a_vowel(s)` with the test procedure called `test_has_a_vowel()`. You are not going to put any tests in the procedure yet, but we do want you to put in a single `print` statement. So right now, your procedure should look like this:

```
def test_has_a_vowel():
    print 'Testing function has_a_vowel()'
```

You should put this procedure definition **below the definition of `test_assert`, but above the script code**. If you put it below the script code then it will not work properly.

The purpose of the `print` statement is so that you have a way to determine whether the test is running properly. Without it, a properly written script will not display anything at all, and we have seen that students find this confusing.

A test procedure is not useful if we do not call it. Add a call to the procedure in the "script code" (e.g. the code the comment `# SCRIPT CODE`). Add the call *before* the final print statement. Once again, run the script `lab04.py`. What do you see?

2.2. **Implement the First Test Case.** In the body of function `test_has_a_vowel()`, you are now going to add several new statements below the `print` statement that do the following:

- Create the string `'aeiou'` and save its name in a variable `s`.
- Call the function `has_a_vowel(s)`, and put the answer in a variable called `result`.
- Call the procedure `cornelltest.assert_equals(True,result)`.

If you want, you can combine all three steps into a single nested function call like

```
cornelltest.assert_equals(True,funcs.has_a_vowel('aeiou'))
```

Either of these approaches will verify that the value of `has_a_vowel('aeiou')` is `True`. If not, it will stop the program and notify you of the problem.

Run the unit test script now. If you have done everything correctly, the script should reach the message `'Module funcs is working correctly.'` If not, then you have actually made an error in the testing program. This can be frustrating, but it happens sometimes. One of the important challenges with debugging is understanding whether the error is in the code or the test.

2.3. **Add More Test Cases for a Complete Test.** Just because one test case worked does not mean that the function is correct. The function `has_a_vowel` can be "true in more than one way". For example, it is true when `s` has just one vowel, like `'a'`. Alternatively, `s` could be `'o'` or `'e'`. We also need to test strings with no vowels. It is possible that the bug in `has_a_vowel` causes it returns `True` all the time. If it does not return False when there are no vowels, it is not correct.

There are a lot of different strings that we could test — infinitely many. The goal is to pick test cases that are *representative*. Every possible input should be similar to, but not exactly the same as, one of the representative tests. For example, if we test one string with no vowels, we are fairly confident that it works for all strings with no vowels. But testing `'aeiou'` is not enough to test all of the possible vowel combinations.

How many representative test cases do you think that you need in order to make sure that the function is correct? Perhaps 6 or 7 or 8? Write down a list of test cases that you think will suffice to assure that the function is correct:

2.4. **Test.** Run the test script. If an error message appears, study the message and where the error occurred to determine what is wrong. While you will be given a line number, that is where the error was *detected*, not where it occured. The error is in `has_a_vowel`.

2.5. **Fix and Repeat.** You now have permission to fix the code in `lab04.py`. Rerun the unit test. Repeat this process (fix, then run) until there are no more error messages.

The Function `replace_first(word,a,b)` (OPTIONAL)

You should have enough experience to work on Assignment 1 now, but we have one more exercise if you want it. Read the specification for the function `replace_first` in `funcs.py`.

In module `lab04.py`, you should make up another test procedure, `test_replace_first()`. Once again, this test procedure should start out with a simple print statement to help you see when it is running, just like you did with `test_has_a_vowel()`. You should also add a call to this test procedure in the script code, before the final print statement.

2.6. **Implement the First Test Case.** This function is different in that your tests now require multiple inputs (not just one). For that reason, we are going to skip the step where you assigned the input to a variable before calling the function. Instead, we will just have you call the function on the inputs directly.

To see what we mean by this, we will get you started with the first test case.

- Call `replace_first` on `'crane'`, `'a'`, and `'o'` and assign the value to `result`.
- Use `assert_equals` to compare result to `'crone'`, the expected value.

In the example above, this input is not just `'crane'`. It is all three values. If you called the function on `'crane'`, `'e'`, and `'k'` (producing `'crank'`), that is actually a separate test case. There should be no error when you run `lab04.py`. Check your test procedure if you run into any problems.

2.7. **Add Another Test Case.** Obviously, that first test case is not enough to test this function. We told you there was an error, and you have not found an error yet. Read the specification for `replace_first`. Why was the first test case not sufficient to test the function `replace_first`?

In the box below, list some better test cases to try out.

Add these test cases to the test procedure `test_replace_first()` and run the unit test script again. You should get an error message now, provided that you chose your test cases correctly.

2.8. **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 `replace_first()` has four assignments. The error could have occurred at any one of them.

We often use `print` statements to help us isolate an error. Recall in class that something as simple as a spelling error can ruin a computation. That is why is always best to *inspect* a variable immediately after you have assigned a value to it.

Open up `funcs.py`. Inside of `replace_first`, after the assignment to `pos`, add the statement

```
print pos
```

Do the same after the remaining three assignments (that is, print `before`, `after`, and `result`). Now run the script. Before you see the error message, you should see four lines printed to the screen. Those are the result of your print statements. These numbers help you "visualize" what is going on in `replace_first()`.

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

2.9. **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.

2.10. **Add Yet Another Test Case.** Guess what? There is a *second* bug with this function. This one is a little more subtle. Read the specification very carefully. Come up with another important testto try. You can tell that it is the test is the correct one if the function *fails* the test.

2.11. **Fix and Test.** The print statements that you put in `replace_first` should still be there, and they should help you identify the error once again. What is it?

2.12. **Clean up `replace_first()`.** Unlike unit tests, using print statements to isolate an error is quite invasive. You do not want those print statements showing information on the screen every time you run the procedure. So once you are sure the program is running correctly, you should 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.

However, once you remove these, it is important that you test the procedure one last time. You want to be sure that you did not delete the wrong line of code by accident. Run the unit test script one last time, and you are done.