CS 1110, LAB 4: ASSIGNMENT 1

http://www.cs.cornell.edu/courses/cs1110/2017fa/labs/lab4/

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 provided a few optional activities to help you along. However, these are activities are **not** necessary to get credit for the lab.

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). That is the only requirement for this lab.

For students who want extra practice, we have provided several optional exercises. These exercises are intended to give you practice with some of the skills that you need for the first assignment. You may work on these optional exercises either directly on this worksheet or through the online system:

http://www.cs.cornell.edu/courses/cs1110/2017fa/labs/lab4/

These optional exercises have no effect on your credit for the lab. Even if you complete them all, you will not receive credit for the lab until you submit Assignment 1.

If you do want to work on the optional exercises today's lab, you will need two files tha are located on the **Labs** section of the course web page.

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). Make sure that both `lab04.py` and `funcs.py` are in this folder before beginning.

Before you modify any of the files, we want you to get use to running a script. Execute `lab04.py` by typing

```
python lab04.py
```

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

Type what you see below.

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** (the comments clearly label which is which). A test procedure is a function that uses the `cornell` 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.

Right now 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 `cornell` 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

```
cornell.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 `cornell.assert_equals` to fail?

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:

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

Run the script one last time and look it what happens.

Based on what you see, how reliable is `assert_equals` on float values?

This should give you some motivation for using the function `assert_floats_equal`. This is a variant of `assert_equals` designed specifically for float arguments.

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

Before you continue on with the part of the lab, make sure that you recomment any of the lines that you uncommented in `lab04.py`. We do not want to (intentionally) be causing errors.

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

Create 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()` (notice the name is similar, but different). 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 after the comment `# SCRIPT CODE`). Add the call *before* the final print statement. When you are done, run the script `lab04.py`

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 value in a variable `s`.
- Call the function `has_a_vowel(s)`, and put the answer in a variable called `result`.
- Call the procedure `cornell.assert_equals(True,result)`.

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

```
cornell.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.

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? Come up with at least 8 different test cases to help assure assure that the function is correct:

Input	**Expected Output**

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`.

Fix and Repeat. You now have permission to fix the code in `funcs.py`. When you think you have fixed the error, rerun the test script. Repeat this process (fix, then run) until there are no more error messages.

3. 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.

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 the 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.

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`?

Give us a better test case reflecting your answer.

Input			Expected Output

Add this test case 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 case correctly.

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?

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.

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 test to try. You can tell that it is the test is the correct one if the function *fails* the test. What is it?

Input			Expected Output

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?

What is the error?

Clean up `replace_first()`. Unlike test cases, 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.