CS 1110, LAB 3: FUNCTIONS AND TESTING

http://www.cs.cornell.edu/courses/cs1110/2015fa/labs/lab03.pdf

First Name: _____ **Last Name**: _____ **NetID**: _____

The purpose of this lab is to help you to better understand functions: both how to write them and how to test them. These concepts are the primary focus of Assignment 1, and therefore it is important that you complete this lab before starting on the assignment.

If you have never programmed before, you might find this lab *significantly* longer than the previous lab. In that case, it is very likely that you will not finish the lab during class time. We ask that you make every effort to complete activities 1 and 2 during the lab time. If you are having any difficulty at all with this lab (especially the first two activities), we strongly encourage you to sign up for one of the **One-on-Ones** announced in class.

Lab Materials. We have created several Python files for this lab. You can download all of the from the Labs section of the course web page.

http://www.cs.cornell.edu/courses/cs1110/2015fa/labs

For today's lab you will notice two files.

- `lab03.py` (a module with your first function)
- `test_lab03.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 `lab03.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.

Getting Credit for the Lab. This lab is unlike the previous two in that it will involve a combination of both code and answering questions on this paper. In particular, you are expected to complete both the module `lab03.py` and the testing script `test_lab03.py`.

When you are done, show all of these (the handout, the test script, and the module) to your instructor. You instructor will then swipe your ID card to record your success. You do not need to submit the paper with your answers, and you do not need to submit the computer files anywhere.

As with the previous lab, if you do not finish during the lab, you have **until the beginning of lab next week to finish it**. You should always do your best to finish during lab hours. Remember that labs are graded on effort, not correctness.

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

1. Writing Your First Function

Up until now, we have only been working with the functions provided by Python. Now it is time to create your own. Recall the very last exercise in the previous lab. In that exercise, you were given a string of the form

```
q1 = 'The phrase, "Don\'t panic!" is frequently uttered by consultants.'
```

You were asked to write a sequence of assignment statements that extracted the substring inside the double quotes. The final answer was stored in a variable called `inner`.

When you checked off the lab, we had you verify that your assignment statements worked on different values of `q1` as well. This got a bit annoying, as you had to type in the assignment statements each time, even though they did not change (only `q1` changed). This is the motivation for writing a function. A function allows us to group all of those assignments together and replace them with a single statement (the *function call*).

Before you write a function, you need to a *module* to store the function. We have already created a module file for you – the file `lab03.py` that you have downloaded for this lab. At the end of this file, you will see the body of a function called `first_inside_quotes()`. It looks like this:

```
def first_inside_quotes(s):
    # Your assignment statements from lab 2 go here

    return inner
```

There is also another function in this file. **Ignore the other function for now**. You should only work on the function `first_inside_quotes()`

The function `first_inside_quotes()` takes a string and returns the substring inside the first pair of double-quote characters. To implement this function, replace the comment with your assignments from the previous lab exercise. However, note that the parameter for this function is `s`, not `q1`. **You must change your assignment statements to use the variable `s` instead of `q1`.**

It is now time to try out your function. Navigate the command line to the folder containing the file `lab03.py` (ask a consultant/instructor for help if you cannot figure out how to do this). Start the Python interactive shell and `import` the module `lab03`. **Remember to omit the `.py` suffix when you use the import command**. Call the function

```
lab03.first_inside_quotes('The instructions say "Dry-clean only".')
```

(Remember the module prefix) What happens?

To check off this portion of the lab you should demo your function to the course staff with a few different arguments. Whenever you call the function, you should make sure that each argument always has a pair of double-quote characters in it, as this is required by the precondition.

2. Working with a Test Script

Now that you know how to write a function, you need to learn how to test it. In the previous step, you tested the 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*.

Recall from class that a script is like a python module, as it is a text file ending in the suffix `.py`. However, we do not import scripts; we run them directly from the command line. For example, the file `test_lab03.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 test_lab03.py
```

This will *not* give you the Python interactive shell with the symbol >>>. Instead, it will run the Python statements in `test_lab03.py` and then immediately quit Python when done. What did the script print to the screen when you ran it?

Now open up `test_lab03.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.

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_float_equals`. 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 `test_lab03.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.

3. Test the Function `has_a_vowel(s)`

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.

3.1. **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()`. 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 indented under `if __name__ ...`). Add the call *before* the final print statement. Once again, run the script `test_lab03.py`. What do you see?

3.2. **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,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 lab03 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.

3.3. **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:

3.4. **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`.

3.5. **Fix and Repeat.** You now have permission to fix the code in `lab03.py`. Rerun the unit test. Repeat this process (fix, then run) until there are no more error messages.

4. Test the Function `first_inside_quotes(s)`

The lab has now come full circle. You started the lab creating your first function. You have also learned how to test a function. It is now time to create a unit test for your function `first_inside_quotes(s)`.

First, you should think of several test cases for `first_inside_quotes(s)`. Come up with at least 4 different test cases, and explain why they are different:

Remember that a test case is both an input **and** and output. We need both

Now that you have your test cases, the process is very much the same as what you did to test `has_a_vowel()` in the previous part of the lab

4.1. **Add a Test Procedure.** In module `test_lab03.py`, you should make up another test procedure, `test_first_inside_quotes()`. Once again, this test procedure should start out with nothing more than a simple print statement indicating that it is working properly. You should also add a call to this test procedure in the script code, before the final print statement.

4.2. **Implement the First Test Case.** Take your first test case from the box above.

- Assign the input to a variable `s`.
- Call `first_inside_quotes` on `s` and assign the value to `result`.
- Use `assert_equals` to verify that `result` is the answer you expected.

4.3. **Test and Fix Errors.** Run the script before you add any more of your test cases. If you get an error, look at your code for `first_inside_quotes(s)` and try to figure out what it is. Keep fixing and testing until there are no errors.

4.4. **Repeat with a New Test Case.** Once you are statisfied that a particular test case is working correctly, start over with the next test case. Continue until there are no test cases left.