CS 1110, LAB 3: FUNCTIONS AND TESTING

http://www.cs.cornell.edu/courses/cs1110/2014fa/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 will 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. If you are having any difficulty at all with this lab, 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/cs1110/2014fa/labs

For today's lab you will notice four files.

- `lab03.py` (a module with your first function)
- `parse.py` (a module with some bugs in it)
- `demo_test.py` (a simple script introduce testing)
- `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.

0.1. **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`. Testing the module `parse.py` is optional.

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 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 paramater 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. Using the `cornelltest` Module

Now that you know how to write a function, you need to learn how to test one. In the previous step, you tested the function by typing a few examples into Python using 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.

For this next part of the lab you will do two things: learn about the module `cornelltest`, and use it in 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 `demo_test.py` for this lab 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 done, type the following:

```
python demo_test.py
```

This will *not* give you the Python interactive shell with the symbol >>>. Instead, it will run the python statements in `demo_test.py` and then immediately quit Python when done.

You will notice that the script displays the help instructions for the module `cornelltest`. Page through this (the spacebar moves to the next page) and look at the functions available.

Now open up the file `demo_test.py` in Komodo Edit and comment out the first print statement (add a `#` at the beginning of that line). Add the followings lines, above the final print statement:

```
cornelltest.assert_equals('b c', 'ab cd'[1:4])
cornelltest.assert_true(3 < 4)
cornelltest.assert_equals(3, 1+2)
cornelltest.assert_equals(3.0, 1.0+2.0)
cornelltest.assert_floats_equal(6.3, 3.1+3.2)
```

Do not indent these lines; they should have the same indentation as the print statements.

Run the script from the command line. Because nothing was received that was not expected you will just get the output `Done with demoing cornelltest`, and nothing else.

Now let us see what happens when something unexpected is received. In the first usage of `assert_equals`, change `'ab cd'[1:4]` to `'ab cd'[1:3]`; then run the script again. This time, you should 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?

Now change the 3 back to a 4 on the first line so that there is no error. In addition, add this line before the final print statement (no indentation):

```
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`:

3. Create a Unit Test Script

Now that you know how to use `cornelltest`, it is time to create a unit test script to check for any errors in the module `lab03`. We have started this unit test for you; it is the file `test_lab03.py`.

This file already has some code in it. In particular, it has the line

```
if __name__ == '__main__':
```

Recall from class that this prevents the the print statement underneath from executing should we accidentally import this script as a module. As a general rule, anything that is not a function definition or a (constant) variable assignment should be indented underneath this line.

Run the script, just like you did `demo_test.py`. What happens?

3.1. **Create a Test Procedure.** The first function in the module `lab03` is `has_a_vowel(s)`. To test this, you are going to create a *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()'
```

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 very 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 print statement. That way, if anything goes wrong in the test procedure, the script will stop before printing the final announcement. 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 result.
- Call the procedure `cornelltest.assert_equals(True,result)`.

If you want, you can combine the last two steps into a nested function call like

```
cornelltest.assert_equals(True,has_a_vowel(s))
```

where `s` is string. This procedure will verify that the value of `has_a_vowel(s)` is `True`. If not, it will stop the program (before reaching the `print` statement) 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 it has just one vowel, such as `'a'`. Similar it can have just `'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 the other ways in which `has_a_vowel()` could be true.

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 unit test script. If an error message appears (e.g. you do not get the final print statement), 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.

5. The Module `parse.py` (OPTIONAL)

This lab is now done; you do not need to do any more to get credit. However, we have provided another module to test, the module `parse.py`. The functions within this module have all have some error. In fact, they are the type of errors that you will likely run into on Assignment 1. Since the consultants are allowed to give you a lot more help on labs than assignments, it is a good idea to try this part of the lab.

First, open the module `parse.py` and look at the specification for the two functions. However, even if you can see the error already, we do not want you to fix it until you are instructed to do so.

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

This function takes a string `word`, and returns a new string where the first instance of letter `a` is replaced with `b`. So `replace_first('crane','a','o')` returns `'crone'`. The precondition always guarantees that `a` is a substring of `word`.

In module `test_lab03.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.

5.1. **Implement the First Test Case.** The test cases for this function will follow a similar format to the previous two unit tests, except that now your tests require multiple inputs (not just one). For that reason, we are going to skip the step from the previous unit tests, 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 actually all three values. If you called the function on `'crane'`, `'e'`, and `'k'` (producing `'crank'`), that is actually a separate test case.

Run the unit test script now. There should not be an error this time; check your test procedure if you run into any problems.

5.2. **Add Some More Test Cases.** 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.

5.3. **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 `parse.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?

5.4. **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.

5.5. **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.

The Function `parse_sum(s)`

Once again, open the module `parse.py` and read the specification for the function `parse_sum(s)`. This function takes a string like `'1+2+3'` and adds the numbers in the string together. Valid strings (e.g. ones that satisfy the precondition) will always have three numbers and two `+` symbols. You do not need to worry about any other types of strings.

As with the two previous problems, add a test procedure called `test_parse_sum()`. You should also add a call to this function in the script code.

5.6. **Implement the First Test Case.** The test cases for this function will follow a similar format to the previous two unit tests, with one minor difference. Inside of the test procedure `test_parse_sum()` you should do the following:

- Assign the string `'1.0+2.1+3.2'` to a variable `s`.
- Call `parse_sum` on `s` and assign it to `result`.
- Use `assert_floats_equal` to compare result to `6.3`, the expected value.

Notice that we ask you to use `assert_floats_equal` instead of `assert_equals`. That is because `parse_sum(s)` returns a float. You saw why this was import when working with `demo_test.py`.

Once this is test case is implemented, run the test script.

5.7. **Oops.** Something different happened. You did not get the nice error message from `assert_equals` this time. Python was not able to complete processing the function and gave you an error that looks like this:

```
File "parse.py", line 43, in parse_sum
     y = float(string2)
ValueError: could not convert string to float:
```

On some computers, the error might instead be `'empty string for float()'`.

In the previous examples, Python just gave you the wrong answer. This time it crashed (And you can tell it crashed because there is no "Quitting with Error"). Unit testing is not going to help you find an error like this. And the line number in the error message is no help either. That is just where Python **found** the error; the mistake could have been made earlier.

Once again, you need to isolate the error with print statements. After **every single assignment statement**, add a print statement displaying the value of the variable from the assignment statement above. Run the unit test script and look at what is displayed on the screen.

This should be enough information for you to find the error. The error here is a legitimate mistake that you might make in a function like this. We made it ourselves when we wrote this function, and then left it in for the lab. If you cannot find the error now, ask a consultant or instructor for help.

5.8. **Fix and Test.** Once you find the error, fix it. Run the test again, and fix it again if necessary. It is a good idea to leave the print statements in until you are sure that the function is correct. However, when it is correct, you should remove all of the print statements inside of `parse_sum()` (and test one last time!).