CS 1133, LAB 3: STRINGS AND TESTING

http://www.cs.cornell.edu/courses/cs1133/2017fa/labs/lab3/

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

The purpose of this lab is to help you to better understand strings and user-defined functions. One of your primary tasks in Assignment 1 will be writing functions that take a string as argument and return a string when done.

This lab will be similar in style to the previous lab. There are several tables that you will need to fill in the values for. These tables will help you understand string methods and expressions.

In addition to the string exercises, we will ask you to create two files. One of them is a module with a function. The other is a testing script. Together, these two activities will take you a long way in understanding how to finish the first assignment.

Because we are trying to give you as much help as possible, this will be a much longer lab than the previous ones. You probably will not finish the lab during section. However, we promise that we will make it up to you. Next week, you will not have a new lab. Instead, you will get a chance to spend lab time on Assignment 1.

Lab Materials. To help cut down on the amount of time that you need to work on this lab, we have started the Python modules/scripts for you already. You just need to download them from the Labs sections of the course web page.

http://www.cs.cornell.edu/courses/cs1133/2017fa/labs

For today's lab you will notice two files.

- `lab03.py` (the module with your function definition)
- `test03.py` (a testing script for `lab03.py`)

You will need to modify both of these files.

You should create a *new* directory on your hard drive and download the files into that directory. You can also get all of the files bundled as a single ZIP file called `lab03.zip`.

Getting Credit for the Lab. Just like the previous lab, you will need to show both this handout *and* your files to get credit. Show all of these to you instructor, who will 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 section, you have **until the beginning of lab next week to finish it**. You can also check of your lab during consulting hours (though this may be hard with Assigment 1 due in 1110 this week). 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. String Expressions

This section should be very familiar to you by now. The first table challenges you to evaluate an expression. The second table asks you to "fill in the blank", completing an expression to give you the provided value. The blanks will all either be a string or number literal.

Throughout this section, pay close attention to spaces and to the different types of quotation marks being used. We use both ' (single quote) and " (double quote). While both work, there are reasons to use one over another. In addition, there is a ` (back quote). While we did not the see the back quote in class, you should be able to figure it out from this lab.

Expression	Calculated
`'CS '+'is '+'fun'`	
`'CS'+'is'+'fun'`	
`"CS"+"is"+"fun"`	
`"CS"+('is'+"fun")`	
`'Double "'`	
`'Single \''`	
`'Single '''`	
`'' + 'ok'`	
`'' + '4 // 2'`	
`'' + 4 // 2`	
`'' + str(4//2)`	

Expression	Calculated	Missing
`'Hello '+` ☐	`'Hello World'`	
`'Hello'+` ☐	`'Hello World'`	
`"Hello "+` ☐	`'Hello World'`	
`"A"+(` ☐ `+"C")`	`'ABC'`	
`'A'+` ☐ `+'B'`	`'A"B'`	
`'A'+` ☐ `+'B'`	`"A'B"`	
`'A'+` ☐ `+'C'`	`"A'B\"C"`	
`''+` ☐	`''`	
`''+` ☐	`'2+2'`	
`'4'+` ☐	`'4 // 2'`	
`str(` ☐ `+ 2)`	`'4'`	

2. String Methods

Strings have many handy methods, whose specifications can be found at the following URL:

http://docs.python.org/3/library/stdtypes.html#string-methods

For right now, look at section 4.7.1 "String Methods," and do not worry that about all the unfamiliar terminology. You will understand it all by the end of the semester.

Using a method is a lot like using a function. The difference is that you first start with the string to operate on, follow it with a period, and then use the name of the method as in a function call.

For example, the following all work in Python:

```
s.index('a')          # assuming the variable s contains a string
'CS 1110'.index('1')  # you can call methods on a literal value
s.strip().index('a')  # s.strip() returns a string, which takes a method
```

Before starting with the table below, enter the following statement into the Python shell:

```
>>> s = 'Hello World!'
```

Once you have done that, use the string stored in `s` to fill out the tables below, just as before.

Expression	Calculated
`s[1]`	
`s[15]`	
`s[1:5]`	
`s[:5]`	
`s[5:]`	
`'e' in s`	
`'x' in s`	
`s.index('e')`	
`s.index('x')`	
`s.index('l',5)`	
`s.find('e')`	
`s.find('x')`	
`s.count('o')`	

Expression	Calculated	Missing
`s[ ]`	`'W'`	
`[ ][2]`	`'C'`	
`s[4: ]`	`'o Wo'`	
`s[: ]`	`'He'`	
`s[ :]`	`'rld!'`	
`[ ] in 'ABC'`	`True`	
`'e' in [ ]`	`False`	
`s.index( )`	`4`	
`[ ].index('x')`	`2`	
`s.index('o', )`	`7`	
`s.find( )`	`4`	
`[ ].find('e')`	`-1`	
`s.count( )`	`3`	

3. Writing a Function on Strings

In the previous lab, you wrote your first function. That function took numbers as its arguments. This time you are going to write a function that requires a string argument.

In the file `lab03.py`, you will find the specification for the function `first_inside_quotes`. This function takes a string containing at least two double-quote characters. It returns the slice from this string that is between the first two double-quotes.

For example, when the function is working properly, you should see the following behavior:

```
>>> import lab03
>>> lab03.first_inside_quotes('A "B C" D')
'B C'
```

However, the function does not work correctly because it is not implemented. It is just a stub (a function that does nothing). Erase the word `pass` and replace it with a proper implementation.

If you are unsure of what to do, review the slides for Lecture 4. There are two examples in that lecture that are very similar to this lab. That is all the guidance we are going to give. However, if you are struggling, please call over a staff member to help you.

4. 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 `test03.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 test03.py
```

This will *not* give you the Python interactive shell with the symbol >>>. Instead, it will run the Python statements in `test03.py` and then immediately quit Python when done. What did the script print to the screen when you ran it?

Now open up `test03.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 contains a collection of **test cases**. The script code contains a call to this test procedure, thereby activating the test cases. The test cases will not be active unless you call the test procedure.

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_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

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

Run `test03.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?

What are the answers to these three questions?

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 the result, explain when you should use `cornell.assert_floats_equal` instead of `cornell.assert_equals`:

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

5. Test the Function `has_a_vowel(s)`

Now that you know how test scripts work, it is time to create test cases to check for 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.

5.1. **Create a Test Procedure.** Following the naming convention showed in class, you should test function `has_a_vowel(s)` with the test procedure called `test_has_a_vowel()`. We have already created this procedure for you. However, right now it is just a print statement with no test cases.

You should notice something strange. This print statement did not appear when you ran the script in the previous section. That is because this test procedure is currently inactive. In fact, we put these print statements in test procedures in order to let us know whether or not they are active.

To make the test procedure active, you will need to call it. For example, you will notice a call to `test_asserts` at the bottom of the file. Add a call to `test_has_a_vowel()` *before* the final print statement. Once again, run the script `test03.py`.

What do you see now?

5.2. **Implement the First Test Case.** In the body of function `test_has_a_vowel()`, you are now going to add several test cases below the `print` statement. To create a test case, you first choose an input to test the function. Before you write any Python, you figure out (in your head) what the correct output should be. Then you write the following two lines of Python:

- Call the function on the input and store it in a variable.
- Use `assert_equals` to compare the correct answer to the one in the variable.

For example, suppose we want to test `has_a_vowel(s)` on the input `'aeiou'`. Then we would write the following lines:

```
result = lab03.has_a_vowel('aeiou')
cornell.assert_equals(True,result)
```

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.

5.3. **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:

5.4. **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`.

5.5. **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.

6. Test the Function `first_inside_quotes(s)`

The lab has now come full circle. You started the lab creating a function on strings. You have also learned how to test a function. It is now time to create a test procedure 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:

Input	Output	Justification

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

6.1. **Add a Test Procedure.** In module `test03.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.

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

- Call `first_inside_quotes` with this input as an argument.
- Assign the value to `result`.
- Use `assert_equals` to verify that `result` is the answer you expected.

6.3. **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.

6.4. **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.