CS 1110, LAB 3: MODULES AND TESTING

http://www.cs.cornell.edu/courses/cs11102013fa/labs/lab03.pdf

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

The purpose of this lab is to help you better understand functions, and to introduce you to the basics of testing. Adopting a strong testing habit is very important for learning programming, particularly for the first assignment. As a warning, we will tell you right now: **the module `funcs` has errors in it. Do not look for them right away**. You should only correct a module when you are told, as we step you through the testing exercise.

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/cs1110/2013fa/labs

When you are done, you should have the following four files.

- `demo_test.py` (a simple script to get started)
- `funcs.py` (the buggy module)
- `parse.py` (another, optional, buggy module)
- `test_funcs.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.

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 the testing script `test_funcs.py` and fix the errors in the module `funcs.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.

1. Using the `cornelltest` Module

For the first 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, in that it is a text file ending in the suffix `.py`. However, we do not import scripts. Instead we run them directly from the command line.

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

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** (ask a consultant/instructor for help if you cannot figure out how to do this), 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(3.0, 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?

The last one is the most tricky. You may see more than one line number in the error message. We will talk about this later in class, but for now you only care about the *first line* number mentioned, the one in file `demo_test_solution.py` (the other line number is in a completely different file).

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`:

2. 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 `funcs`. We have started this unit test for you; it is the file `test_funcs.py`.

This file already has some code in it. In partiular, 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 variable assignment should be indented underneath this line.

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

2.1. **The Class `Point`.** We introduced the type Point in lecture; objects of type Point are points in 3-dimensional space. They have three attributes, `x`, `y`, and `z`, corresponding to the three spatial coordinates, stored as floats. You create Point objects with a constructor call, supplying three arguments to set the coordinates `x`, `y`, and `z`. For example, the constructor call

```
Point(2,1,0)
```

creates a Point object with (x,y,z) = (2.0, 1.0, 0.0) and returns the id of the object (what is written on the folder tab on the left).

To use the class Point, you must `import` the module `tuple3d`. Therefore you have to preface the constructor call above with the prefix `tuple3d`, as you would for any function in the module.

2.2. **Create a Test Procedure.** The first function in the module `funcs` is `has_a_zero(p)`. To test this, you are going to create a *test procedure* called `test_has_a_zero()`. Right now, this procedure should just be a "stub" (e.g. it should not do anything at all). To make a stub procedure, just put `pass` indented under the header. So the procedure should look like this:

```
def test_has_a_zero():
    pass
```

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 out the final announcement.

2.3. **Implement the First Test Case.** In the body of function `test_has_a_zero()`, delete `pass` and replace it with Python statements that do the following:

- Create a `Point` object (0,0,0) and save its name in a variable `p`.
  Remember that the constructor function `Point(x,y,z)` takes three arguments.
- Call the function `has_a_zero(p)`, 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_zero(p))
```

where `p` is a variable that contains the (name of) the point object. The procedure `assert_true` will check if the value 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 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.

2.4. **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_zero()` can be "true in more than one way". For example, it is true when x is 0, but none of the other coordinates are. Similarly it can be true when just y is 0, or when just z is zero.

We also need to test points that have no zeroes in them. It is possible that the bug in `has_a_zero()` is that it returns `True` all the time. If it does not return False when the point has no zeroes, it is not working either.

There are a lot of different points 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 point with no zeroes, we are fairly confident that it works for all points with no zeroes. But testing (0,0,0) is not enough to test the other ways in which `has_a_zero()` 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:

2.5. **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_zero()`.

2.6. **Fix and Repeat.** You now have permission to fix the code in `funcs.py`. However, you should restrict your fixes to the function `has_a_zero(p)` only, as this is the only thing that you are testing. Do not fix the other function yet.

Rerun the unit test. Repeat this process (fix, then run) until there are no more error messages.

---

3. Test the Procedure `cycle_left(p)`

The function `cycle_left()` is actually a procedure. It does not return anything. Instead, this procedure changes the contents of the object (e.g. the folder) whose name is in `p`. Read the specifications of this procedure to understand what it does.

In module testfuncs.py, you should make up another test procedure, `test_cycle_left()`. Once again, this test procedure should start out as a stub; put `pass` under the header as you did with `test_has_a_zero()`. You should also add a call to this test procedure in the script code, before the final print statement.

3.1. **Implement the First Test Case.** This procedure should take a point, and "shift" all of the coordinates to the left (with the x coordinate moving to the z coordinate). To test this out, you need to add the following code to `test_cycle_left()`.

- Create a `Point` object (0,0,1) and save its name in a variable `p`.
- Call the procedure `cycle_left(p)`.
- Test that `p` is now the point (0,1,0).

The last step requires further details. You cannot write

```
p == (0,1,0)
```

This will return `False`. That is because (0,1,0) is not a Point object. It is a value of a type that we have not yet seen in class (and will not see for a while). Instead, you have to check each of the attributes x, y, and z separately.

Remember that the attributes of `p` are all floats. Therefore, you want to use the function `assert_floats_equal()` to check that the values are all correct. So, to check that `p` is the point (0,1,0), you would add the following statements:

```
cornelltest.assert_floats_equal(0,p.x)
cornelltest.assert_floats_equal(1,p.y)
cornelltest.assert_floats_equal(0,p.z)
```

Add these test cases to the test procedure `test_cycle_left()` and run the unit test script. There should not be an error this time; check your test procedure if you run into any problems.

3.2. **Add More Test Cases for a Complete Test.** Obviously, the point (0,1,0) is not enough to test this function. We told you there was an error, and you have not found an error yet. Why is this point not sufficient to test the function shift?

What are good points for testing out this function?

Implement the test case(s) above, and run the script again. You should get an error message now.

3.3. **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 `cycle_left()` has three lines of code. 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 `cycle_left`, after the assignment to `p.x`, add the statement

```
print p.x
```

Do the same after the remaining two assignments (that is, print `p.y` and `p.z`). Now run the script. Before you see the error message, you should see three numbers print out. Those are the result of your print statements. These numbers help you "visualize" what is going on in `cycle_left()`.

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

3.4. **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.

3.5. **Clean up `cycle_left()`.** 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.

4. The Function `parse_point()` (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 function within this module has a special type of error; one 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 might be a good idea to try this part of the lab if you run into trouble.

First, open the module `parse.py` and read the specification for the function `parse_point()`. This function takes a string like `'(1,2,3)'` and turns it into the equivalent Point object. You should try to understand this function thoroughly, as it uses techniques found in the first assignment.

As with the two previous problems, add a stub for a test procedure called `test_parse_point()`. In addition, put a call to this function in the script code.

4.1. **Implement the First Test Case.** Testing this function is very similar to testing `cycle_left(p)`. The primary difference is that `parse_point()` is a function that returns a new Point, not a procedure that modifies an existing Point. So your test cases should be testing the point that parse(s) returns, not the string you pass to it.

For example, your first test case should do the following:

- Call `parse_point('(1,2,3)')` and assign the result to a variable `p`
- Test that `p` is now the point (1,2,3).

Follow the steps from `test_cycle_left()` for the second step.

4.2. **Oops.** Something different happened. You did not get the nice `AssertError` message that you normally get from using `cornelltest`. Instead, you got a different error that looks like this.

```
File "parse.py", line 45, in parse_point
     p.y = float(ystring)
ValueError: could not convert string to float:
```

If you get something other than an `AssertError`, that means the Python crashed before it finished evaluating the function. So something inside of `parse_point()` is causing it to crash.

Unit testing is not going to help you find an error like this. Your program is crashing before it can evaluate `assert_floats_equal()` Furthermore, the line number in the error message is no help either. That is just where Python found the error; the mistake actually occurs much earlier in the function.

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.

4.3. **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_point()` (and test one last time!).