CS 1110, LAB 3: FUNCTIONS AND TESTING

Name: _____ **Net-ID**: _____

There is an online version of these instructions at

http://www.cs.cornell.edu/courses/cs1110/2012fa/labs/lab3

You may wish to use that version of the instructions.

The purpose of this lab is to get you used to writing functions, and to introduce you to the basics of testing. As a warning, we will tell you right now: **The module `pointfuncs` has errors in it; do not look for them and test them in the beginning**. You should only correct the module when you are told. The point of this lab is get you in the habit of testing your programs. Adopting this testing habit will prove to be unbelievably useful, particularly for the first assignment.

Requirements For This Lab. We have created a few files for this lab. You should create a new directory on your hard drive and download the following modules into that directory:

- `point.py` (http://www.cs.cornell.edu/courses/cs1110/2012fa/labs/lab3/point.py)
- `pointfuncs.py` (http://www.cs.cornell.edu/courses/cs1110/2012fa/labs/lab3/pointfuncs.py)
- `cunittest.py` (http://www.cs.cornell.edu/courses/cs1110/2012fa/labs/lab3/cunittest.py)
- `testfuncs.py` (http://www.cs.cornell.edu/courses/cs1110/2012fa/labs/lab3/testfuncs.py)

The first two modules are the code that you will be testing. The module `point` provides a new type, called Point. We describe this type below, but you do not need to understand the contents of this file. The most important module is `pointfuncs`. This is the module that has mistakes, and you will (eventually) need to fix them.

The second two modules are used for testing. You will find that the module `cunittest` will be a valuable tool for you throughout the semester. This module contains the unit testing functions `assert_equals` and `assert_true` that we showed off in class. The second is a skeleton module (e.g. there is really nothing in it); it is where you will write your unit tests to find the errors in `pointfuncs`.

This lab will involve three different components. First, you will need to write answers to the questions that we pose below. Second, you will need to write a new module that is a unit test for `pointfuncs`. Finally, you will need to modify the contents of `pointfuncs` to fix the mistakes. When you are done, you should show *all three* to your lab instructor, who will record that you did it. You do not need to submit the paper with your answers, and you do not need to submit the computer files anywhere.

As always, 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. The Module `point`

The module `point` provides a new type: Point. While this type is important for this lab, the contents of this module are not. In fact, you will not learn how to read a module like this until much later in the course. Instead, we describe this type here.

Objects of type Point are points in a 3-dimensional space. These objects have three attributes: `x`, `y`, and `z`. These attributes correspond to the 3 coordinates. Point objects do not have any methods that are relevant to this lab.

Like the Window type in the previous lab, you create Point objects with a constructor. The constructor `Point` takes three arguments to set the coordinates `x`, `y`, and `z`. For example, the constructor call

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

creates a Point object (2,1,0) and returns the name of the object.

---

2. Creating a Unit Test

For the first part of the lab, you will create a unit test to test the functions of module `point-funcs`. You will start by testing the procedure `shift(p)`. You are to continue testing, until you get no error messages. In this section we take you through this process, step-by-step.

2.1. **Create the Unit Test Module.** For the first part of the lab, you will create a unit test to test the functions of module `pointfuncs`. You will start by testing the procedure `has_a_zero`. This function should return true if at least one of the x, y, or z coordinates in the point p is 0. If none of them are zero, it returns false.

That is what the specification says, but the function has a bug and does not work correctly. You are to test the program to find the bug. Some of you may see the error right away, but **do not fix it**. The purpose of this lab is to teach you testing. So we are going to take you through this process, step-by-step.

2.2. **The Unit Test Module `testfuncs`.** A unit test is a special module that is used to test other modules. We have provided you a file to get started – `testfuncs.py` – but it does not have any significant code in it yet. It just has the initial comments at the top of the module (and you should put your name in the right comment). It also has some import statements.

The file also imports `cunittest`. This module provides the functions `assert_equals` and `assert_true` which you will use in unit testing. You will note that we have used the normal `import` keyword to import it, so all calls of those functions will need `cunittest` in front of them (e.g., `cunittest.assert_equals(...)`) to work. We did this just to give you experience with both versions of import, not because it is necessary.

When you create future unit tests in this class (such as for the first assignment), they should start out very much like this skeleton. You need to import the module `cunittest` to use the testing functions, which means you will need a copy of the file `cunittest.py` in the directory that contains the code you are testing. In addition, you obviously need to import whatever module you are testing.

2.3. **The Application Code.** Unit tests are *applications*. An application is a Python module that executes when you type

```
python module-name
```

from the command shell. You can also run an application by having its window as the active window in Komodo Edit and pressing the "run button."

An application ends in a very special bit of code that starts

```
if __name__ == "__main__":
```

The code that executes when you run the application is indented directly under this line, like in a function body. Right now, the application code should contain a single print statement, as follows:

```
if __name__ == "__main__":
    print "Module point-funcs is working correctly"
```

Run the application. What happens?

2.4. **Create the Test Procedure.** You are going to create a procedure `test_has_a_zero()` which will test the procedure `has_a_zero(p)`. 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 right now, this procedure should look like this:

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

Add a call to the procedure in the "application code" (e.g. the code indented under `if __name__` ...). Add the call *before* the print statement. The idea is that, if anything goes wrong in this test procedure, the program will stop before printing out the final announcement.

2.5. **Implement the Test Cases.** In the body of function `test_has_a_zero`, write Python statements that do the following:

- Create a `Point` object (0,0,0), using the constructor `Point`, and store the (name of the) object in a variable `p`.
- Call `has_a_zero(p)` and put the result in a variable named `result`.
- Call the procedure `cunittest.assert_true(result)`.

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

```
cunittest.assert_true(has_a_zero(p))
```

where `p` is a variable that contains the (name of) the point object. The important thing here is that `assert_true` does nothing if the call `has_a_zero(p)` returns True, which it should because your point has all zeros. If anything is wrong, then the `assert_true` function will stop the entire program and notify you of the error.

You should run the unit test now. If you have done everything correctly, then the unit test should reach the message `"Module pointfuncs 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 tester or the testee.

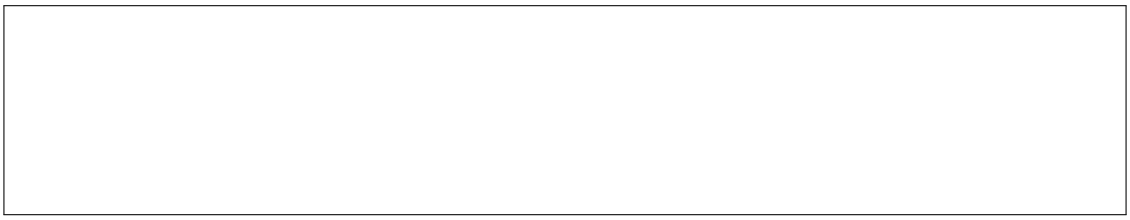
2.6. **Add More Test Cases for a Complete Test.** Just because one test case worked does not mean that the function is correct. The function `has_a_zero` can be "true in more than one way". For example, it

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 0.

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.

Clearly, there are a lot of different points that we could test – effectively infinitely many. The idea 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 `test_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:



In test procedure `test_has_a_zero()`, Implement all these test cases in procedure `test_has_a_zero()`, using the `assert_true` function. If you want to test that something is False, use the **not** operator to make the expression True so that you can use `assert_true` The test procedure may have to create more than one instance of type `Point` in order to implement all of your test cases.

2.7. **Test.** Run the Python module `testfuncs.py` as an application. If an error message appears (so you do not get the final print statement), study the message and where the error occurred (you will be provided with a line number) to determine what is wrong. The error could be anywhere.

2.8. **Fix and Repeat.** You now have permission to fix the code in `pointfuncs.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 anything else yet.

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

---

3. Test Function `shift(p)`

The function `shift(p)` is actually a procedure. It does not return anything. Instead, this procedure will change 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. Testing this will be a little different from testing `has_a_zero`.

In module `testfuncs.py`, you should make up another test procedure, `test_shift()`, that will test the function `shift(p)`. Make this procedure a stub for now. You should also add a call to this test procedure in the application 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_shift`.

- Create a `Point` object (0,0,1), using the constructor `Point`, and store the (name of the) object in a variable `p`.
- Call the procedure `shift(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 a value of a type that we have not yet seen in class (and will not see for a while). Technically, it is correct to use the test

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

However, as we will see later when we cover Classes, this dangerous and does not always work; you just happen to be lucky that it would work in this specific case. Instead, we would prefer that you check each of the attributes – x, y, and z – separately.

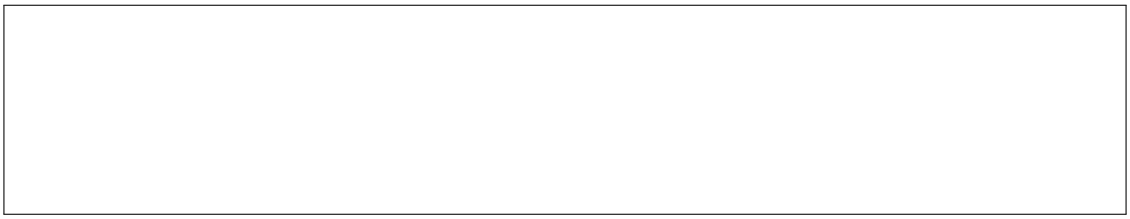
This time you are testing variables with int or float values, not just boolean. To test this type of value you need the function `assert_equals`. In `assert_equals`, you have a value that you expect which you compare against the value that you actually get. So to check that `p` is the point (0,1,0), you would add the following statements:

```
cunittest.assert_equals(0,p.x)
cunittest.assert_equals(1,p.y)
cunittest.assert_equals(0,p.z)
```

Add these test cases to the test procedure `test_shift` and run the unit test as an application. 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) you chose, and run the unit test as an application. You should get an error message this time.

3.3. **Isolate the Error.** Unit tests are great at finding whether or not an error exists. They are not always great at telling you where the error occurred. The procedure `shift` has three lines of code. The error could have occurred at any one of them.

In programming, we often use print statements to help us isolate an error. Recall in the last lab that you were asked to write a sequence of assignment statements where you extracted a substring contained in quotes. If we ran into problems, we suggested that after every print statement you put

```
print var
```

where *var* is the name of the variable in the assignment statement just above it. This will help you "visualize" what is going on. Everytime a variable is created or changes value, it is important that the new value is what you expect it to be.

Open up `pointfuncs` and add these print statements to the function `shift(p)` (not `test_shift()`), one after each of the three assignment statements. Now run the unit test. Before you see the error message, you should see three numbers print out. Those are the result of your print statements.

To make them easier to understand, sometimes we like to add more information to the print statement, such as

```
print "The variable p.x is "+`p.x`
```

If you want to make that change, fine. You should do whichever you are most comfortable with.

3.4. **Fix and Test.** You should now have enough information from these three print statements to see what the error is. Fix the error and test the procedure again by running the unit test.

3.5. **Clean up `shift(p)`.** 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 them. 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 accidentally delete the wrong line of code by accident.

Once you have removed all the print statements, and the unit test runs without errors, you are done with this procedure.

---

4. Test Function `parse(s)`

Read the specification for `parse`. 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 is very relevant to the first assignment.

Create a stub for a test procedure called `test_parse()` and add a call to it to the application code, just like you have done for the prior two parts of the lab.

4.1. **Implement the First Test Case.** Testing this function is very similar to testing `shift(p)`. The primary difference is that `parse(s)` 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. So your first test case should do the following.

- Call `parse("(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_shift()` for the second step.

4.2. **Oops.** Something bad 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 "pointfuncs.py", line 65, in parse
    p.y = float(ystring)
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 in the assignment statement above it. Run the unit test 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` (and test one last time!).

---

5. Add the Function `first_inside_quotes(s)`

You may notice that `pointfuncs` has a specification for a function called `first_inside_quotes(s)`. If a string `s` contains at least two double quotes inside of it, this function returns the substring within the first pair of such quotes.

This might seem familiar. We asked you to do something like this on the last lab. The only difference is that now we are asking you to write it in function form. You will find this function incredibly useful for the first assignment.

Once you are done implementing the function, add one last test procedure: `test_first_inside_quotes()`. Make sure that your function is correct. When you are satisfied, you are done with the lab.