CS 1110, LAB 03: OBJECTS, FUNCTIONS, TESTING

http://www.cs.cornell.edu/courses/cs1110/2013sp/labs/lab03/lab03.pdf

D. GRIES, L. LEE, S. MARSCHNER, AND W. WHITE

First Name: ________________ **Last Name**: ________________ **NetID**: ____________

Relevant lecture material. The handouts, slides, and code examples from last Tuesday and Thursday's lectures and the handout for this Tuesday's lecture are at http://www.cs.cornell.edu/courses/cs1110/2013sp/lectures/ .

Files to download. Create a *new* directory on your hard drive and download the following modules into that directory. (You can get them all bundled in a single zip file at http://www.cs.cornell.edu/courses/cs1110/2013sp/labs/lab03/lab03files.zip)

`cunittest.py` (http://www.cs.cornell.edu/courses/cs1110/2013sp/labs/lab03/cunittest.py)
`democunittest.py` (http://www.cs.cornell.edu/courses/cs1110/2013sp/labs/lab03/democunittest.py)
`point.py` (http://www.cs.cornell.edu/courses/cs1110/2013sp/labs/lab03/point.py)
`pointfuncs.py` (http://www.cs.cornell.edu/courses/cs1110/2013sp/labs/lab03/pointfuncs.py)
`testpointfuncs.py` (http://www.cs.cornell.edu/courses/cs1110/2013sp/labs/lab03/testpointfuncs.py)

Getting credit for lab completion. When done, show your code and/or this handout to a staff member, who will ask you a few questions to see that you understood the material and then swipe your ID card to record your success.

As always, if you do not finish during the lab, you have until the beginning of lab next week to finish it: show it to your lab TA at the beginning of that next lab. But you should always do your best to finish during lab hours. Remember that labs are graded on effort, not correctness.

---

1. The Module `cunittest`

This module contains "convenience" functions that report informative output for testing purposes. Let's practice with it to understand what its functions do.

First, open `democunittest.py` in Komodo Edit, read it, and then do "Run Python Module". In the Command Output pane, you'll see the docstring specifications of all the functions in `cunittest`. Scan this information — you'll probably have to scroll up and/or resize the Command Output pane — to get a quick sense of what these "assert"-style functions are supposed to do.[1]

(Turn this page over)

[1]You could also have seen this info by reading the file `cunittest.py`, but we want you to know that you can always call (for) `help`.

Now, comment out the first print statement (by adding a # at the beginning of that line), and then put the following lines into the file, above the final `print` statement.[2] Don't indent them.

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

Do "Run Python Module", and you'll see that because nothing was received that wasn't expected, you just get the output `Done with demoing cunittest`.

Now let's see what happens when something unexpected is received. Change `'ab cd'[1:4]` to `'ab cd'[1:3]`, and do "Run Python Module". In the Command Output pane you'll see *three important pieces of debugging information* (in this case, occurring in the first two lines of the output):

- what was (supposedly) expected
- what was received
- which line caused `cunittest.assert_equals` to fail

See that you understand these three pieces of output; then, change the 3 back to a 4.

Finally, add this line before the final print statement (no indentation):
`cunittest.assert_equals(6.3, 3.1+3.2)`
and do "Run Python Module".

Given this last experiment, explain when one should use `cunittest.assert_floats_equal` instead of `cunittest.assert_equals`:

---

2. The Module `point`

You saw type Point in lecture. Reminder: 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's written on the folder tab on the left, in our folder notation).

---

[2]We're asking you to type these in, rather than giving these lines to you already in the file, because any typos you make will help you further understand how these functions work.

3. Testing a Function

We are going to test the procedure `cycleleft` that has been placed in module `pointfuncs`. This procedure[3] is intended to alter the coordinates of the point object it is given as an argument, in the way explained in its specification.

Open `pointfuncs.py` in Komodo Edit and read the specification of `cycleleft` carefully.

Next, open `testpointfuncs.py`. This *unit test* is the module in which you will be placing testing code. Right now it just contains some template code of the form discussed in lecture. Correct the authoring-info header and proceed.

3.1. Stub in the Test Procedure and Put It in the Application Code.

Create a procedure `testCycleleft()` in module `testpointfuncs` that tests procedure `cycleleft(p)` in `pointfuncs`. First, "stub" it in: just put in the minimum skeleton or framework needed for your test procedure to exist, namely:

```
def testCycleleft():
    pass # stub
```

(Congratulations, by the way; you've just written your first Python function! The "pass" statement does nothing in particular, but functions need bodies.)

Then, add a call to `testCycleleft()` in the "application code"(i.e., under `if __name__` ...), before its print statement. The idea is that if anything goes wrong in your test, the program will stop with an error message before printing out the final announcement.

3.2. Implement a Test Case.

In the body of function `testCycleleft`, write Python statements that do the following:

- Create a `Point` object with (x,y,z) = (0, 0, 1) and save its id in a variable `p`.
- Call the procedure `pointfuncs.cycleleft` on `p`.
- Insert three tests (assert statements) to verify individually that after the call, the new values of attributes `p.x`, `p.y`, and `p.z` are correct.[4]
- Remove the pass statement, since your procedure now has a real body.

If you need an example to look at, you can consult module `testgym`, `http://www.cs.cornell.edu/courses/cs1110/2013sp/lectures/01-31-13/testgym.py`, for the general format.

Now, do "Run Python Module". If any errors occur, they have probably occurred due to a problem in your testing code; fix them before moving on.

3.3. Add More Test Cases to Test More Thoroughly.

You are surely aware that testing a single point isn't enough. What are some other good test cases? Write two or three down here: you should be specifying an input point and the expected output of `pointfuncs.cycleleft` (that is, you don't have to write code).

(One more page to go!)

[3] A procedure is a function without a `return` statement, in contrast to what your text calls fruitful functions, which do return a value.

[4] Do *not* try to "directly" test that `p` is "equal to" (0, 1, 0) or to a new Point point.Point(0,1,0); as we will see later in the course, this poses certain dangers to the uninitiated.

Now, code up your test case(s) as Python statements in `testCycleleft` and do "Run Python Module" again. You should discover that there is an error in `cycleleft`.

3.4. **Isolate the Error.** Although unit tests can tell you that an error exists, they may not provide enough information to determine what the error is or where it occurred. In this case, we need to look into what values are being stored where.

Each line of `pointfuncs.cycleleft` assigns a value to a variable, and thus has the form "var = val". We need to check whether we're putting the right value in the right place. We can do this with `print` statements. Before each "var = val" line, add a line of the form:

```
print 'var is ' + str(var) + ' and we are changing it to val, which is ' + str(val)
```

For example,

```
print 'p.x is ' + str(p.x) + ' and we are changing it to p.y, which is ' + str(p.y)
```

Save `pointfuncs`, and now run the unit test `testpointfuncs`. Before you see the error message, you should see your requested print output.

You should now have enough information from these print statements to see what the error is. What is it?

Optional: if you have time, answer the following question: How should the error be fixed?

After getting checked in, if you have the time and inclination (it *is* good practice), try fixing the code (changing your print comments appropriately) and re-running your tests, to see if you were right. If you were, remove or comment out your print statements, to make `cycleleft` readable again.