Python Exercise 1

1. **Working with a Test Script**

In the previous exercises, you tested your 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*

A script looks like a Python module, except that we do not import scripts. We run them directly from the command line. The file lab01.py is a script. To run this file, **navigate the command line to the folder with this file**, but do not start Python (yet). Make sure that both lab01.py and funcs.py are in this folder before beginning.

Before you modify any of the files, we want you to get use to running a script. Execute lab01.py

by typing

**python lab01.py**

Type what you see below.

Testing the introcs asserts

Module funcs is working correctly

Now open up lab01.py in Sublime Text. As with the test script in class you will notice two things: a **test procedure** and the **script code** (the comments clearly label which is which). A test procedure is a function that uses the introcs’s module to test *other* functions. The script code contains a call to this test procedure, as the procedure cannot do any testing if you do not call it.

Right now 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 introcs module. In par- ticular, you will see the three functions assert\_equals, assert\_true, and assert\_floats\_equal.

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

introcs.assert\_equals("b c", ab cd"[1:3])

Run lab01.py as a script again. You will see answers to *three important debugging questions*:

* What was (supposedly) expected?
* What was received?
* Which line caused introcs.assert\_equals to fail?
* The expected output was 'b c'
* The received output was 'b '
* Line 18 caused the function to fail: introcs.assert\_equals('b c', 'ab cd'[1:3])

Run the script one last time and see what happen

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:

introcs.assert\_equals(6.3, 3.1+3.2)

Based on what you see, how reliable is assert\_equals on float values?

Not reliable for float comparison due to limited floating point precision.

This should give you some motivation for using the function assert\_floats\_equal. This is a variant of assert\_equals designed specifically for float arguments.

1. **The Function has\_a\_vowel(s)**

Before you continue on with the part of the lab, make sure that you recomment any of the lines that you uncommented in lab01.py. We do not want to (intentionally) be causing errors.

Now that you know how test scripts work, it is time to create a unit test procedure to check for any errors in the module funcs. You are going to start by testing the function has\_a\_vowel(s). We guarantee that this function has a bug in it.

**Create a Test Procedure.** Following the naming convention showed in class, you should test has\_a\_vowel(s) with the test procedure called test\_has\_a\_vowel() (notice the name is similar, but different). 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()')

You should put this procedure definition **below the definition of** test\_assert**, but above the script code**. If you put it below the script code then it will not work properly.

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 useful if we do not call it. Add a call to the procedure in the “script code” (e.g. the code after the comment # SCRIPT CODE). Add the call *before* the final print statement. When you are done, run the script lab01.py.

**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 value in a variable s.
* Call the function has\_a\_vowel(s), and put the answer in a variable called result.
* Call the procedure introcs.assert\_equals(True,result).

If you want, you can combine all three steps into a single nested function call like

introcs.assert\_equals(True,funcs.has\_a\_vowel('aeiou'))

Either of these approaches will verify that the value of has\_a\_vowel('aeiou') is True. If not, it will stop the program 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.

**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? Come up with at least 8 different test cases to help assure assure that the function is correct:

|  |  |
| --- | --- |
| **Input** | **Expected Output** |
| ‘alicE’ | True |
| ‘huhu’ | True |
| ‘’ | False |
| ‘ccmk’ | False |
| ‘z Oo’ | True |
| ‘aaa AaAa aaaaa’ | True |
| ‘mGaNgA’ | True |
| ‘gILdUr’ | True |

**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.

**Fix and Repeat.** You now have permission to fix the code in funcs.py. When you think you have fixed the error, rerun the test script. Repeat this process (fix, then run) until there are no more error messages.

1. **The Function replace\_first(word,a,b)**

You should have some experience functions, strings, testing, but we have one more exercise if you want it. Read the specification for the function replace\_first in funcs.py.

In module lab01.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.

**Implement the First Test Case.** This function is different in that your tests now require multiple inputs (not just one). For that reason, we are going to skip the step 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 the result to 'crone', the expected value.

In the example above, this input is not just 'crane'. It is all three values. If you called the function on 'crane', 'e', and 'k' (producing 'crank'), that is actually a separate test case. There should be no error when you run lab01.py. Check your test procedure if you run into any problems.

**Add Another Test Case.** 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?

Because the provided test case doesn't check edge conditions.

Give us a better test case reflecting your answer.

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | | | **Expected Output** |
| ‘diana’ | ‘a’ | ‘w’ | ‘diwna’ |

Add this test case 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 case correctly.

**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 funcs.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?

All variableswas printed alongside the description string for each ones in **replace\_first().**

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

The function **replace\_first()** returns a copy of word with the LAST instance of string **a** replaced by string **b.**

Fix the error and test the procedure again by running the unit test script.

**Add Yet Another Test Case.** Guess what? There is a *second* bug with this function. This one is a little more subtle. Read the specification very carefully. Come up with another important test to try. You can tell that it is the test is the correct one if the function *fails* the test. What is it?

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | | | **Expected Output** |
| ‘trung’ | ‘f’ | ‘o’ | ‘trung’ |

**Fix and Test.** The print statements that you put in replace\_first should still be there, and they should help you identify the error once again. What is it?

What is the error?

In the test case of string **b** not appearing in **word**, the result is different from the expected original string output.

**Clean up** replace\_first()**.** Unlike test cases, 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.