CSC120H Lab 10
Testing Lab

As a chance to try something different (and something very important), we're going to spend most of the lab learning how to use Python to test your functions in a more efficient way. This is important if you will be writing large programs with many functions, and so it is good to see it before the end of the course.
The last section of this lab will have some more SQL query practise with aggregate functions - for this SQL section of the lab, the solutions will also be posted on the Labs page for you to try and check, either if you have time in the lab, or at home.

1 Objectives

- Practise selecting good test cases
- Practise implementing a set of test cases using `unittest`.

2 Testing using `unittest`

Throughout the term, we have emphasized the importance of testing your code on different types of inputs to ensure that your functions are written correctly. You've written 2 or 3 docstring examples per function which indicate what your function is supposed to return given a certain set of arguments.
However, what happens when you want to test more than just a few cases? It's tedious to have to write lots of examples in the docstring, and can take time to copy them all into the shell.

Thankfully, Python has a module you can import called `unittest`. This module gives you a way to define as many example cases as you want in a separate file, and then run them all at once to see what's working, and what you might need to fix!

You will be writing `unittest`s to thoroughly test some functions, and then use those tests to identify and fix any bugs that you find.
Let's see how it works:

1. Download `buggy_functions.py` and `test_is_lowercase.py` from the Labs page.
   The module `buggy_functions.py` contains the function `is_lowercase` as well as a handful of other functions for which you will write tests.
   `test_is_lowercase.py` contains a set of `unittest` tests for `is_lowercase`. Take a look at the structure of `test_is_lowercase.py`. There are some interesting things here:

   `class TestIsLowercase(unittest.TestCase):`
   The `class` keyword defines the beginning of the structure for our set of test cases. You don't have to worry about what it means in other contexts right now.
   `TestIsLowercase` is the name of our set of tests, in this case, the tests that are going to test the function `is_lowercase`.

   Below this line, there are a bunch of functions that define the **test cases** we want to test. Each of the tests contains a single method call that checks that the function being tested returns the correct value for the given argument (method `assertTrue` checks whether its first argument is `True` and `assertFalse` checks whether its first argument is `False`). Each test is named to inform you that it is a test for a specific call to `is_lowercase` (for example, `test_empty` tests the result of calling the function on an empty string). You do not have to use the `self` parameter in your code.
   Read the comments in the first test and ensure you understand what they mean.
   The line of code at the bottom of the file, `unittest.main(exit=False)`, causes all of the test functions in the module to be executed.

2. Run `test_is_lowercase.py`.
   In the shell, you will see the 'test output', which tells you which tests passed, and which tests failed.
   Don't be alarmed to see Python errors in the shell, they won't actually stop all of the tests from running - they will just show up as failed tests!
   There are two possible test case failure types:
   - `E` or `ERROR`, which means that an error occured while running your code (this could be a syntax error, or a list index out of range error).
   - `F` or `FAILURE`, which means that your function finished, but the return value was not the one that was expected.
   For example, the test `test_empty` was a `FAILURE`, because it was supposed to return True, but returned False instead. This is because of the buggy if-statement at the beginning of the function in `buggy_functions`.
   The bottom of the shell output show the total number of errors and failures: `FAILED (failures=1, errors=2)`

   Task: Comment out the buggy if-statement in `is_lowercase` to fix the code. After changing the function, notice how easy it is to just **re-run the tests** by running `test_is_lowercase.py` to verify that all tests now pass - you don't have to put examples into the shell yourself!
   This particular test (the empty string test) should now pass. However...
   ..you'll find some other tests for `is_lowercase` that encountered an ERROR.

   Task: Look at the shell output to determine which cases are triggering the errors and then edit `buggy_functions.py` to correct them.

   Task: There are two empty tests: `test_longer_non_alphabetic` and `test_longer_uppercase`.
   Create a test case for each of those functions (in a similar format as the other ones). Include an appropriate message in the `msg` variable. Run your tests to ensure they all pass.

   Continued...

3. The table below describes some other functions in the `buggy_functions.py`. Several (perhaps all) of the functions in `buggy_functions.py` have at least one error. Your job is to find them.

`evens`: `(list) -> list`	Return a new list consisting of those members of the given list whose index is even. Items in the new list should be in the same order as they appear in the original list. Do not modify the original list.
`reverse`: `(list) -> list`	Return a new list that contains the items from the given list in reverse order. Do not modify the original list, and do not use `list.reverse`.
`left_strip`: `(str, str) -> str`	Given two strings where the second is a single-character string, return a new string identical to the first string but with any occurrences of the second string removed from the beginning. For example, `left_strip('fffabcdefffg', 'f')` should return `'abcdefffg'`. (This is much like `str.lstrip`.) Do not use any `str` methods to implement this function.

Use the following steps for each of the functions:

(a) For each function, download the corresponding testing module from the Labs page.
Look at the `unittest` test methods that have been written for you in the file, and understand what they do (some of the provided tests pass despite the buggy functions). Each of the tests use `assertEqual` to check that the value produced by the function is the expected value. `assertEqual` takes either two or three parameters: the first is the actual value produced by the function, the second is the expected value, and the (optional) third parameter is a descriptive message to be printed iff the test fails. This lets you compare the actual output from the function with the output that you expect the function to have produced.

Task: Write some more `unittest` test methods in each of the test files. You should write some that pass and some that **fail** (the functions are buggy, after all!).
Include this descriptive message for each `assertEqual` call.

(b) Run your tests. If you find errors, debug and correct the errors making as **little change** to the code as possible. **Do not simply delete and rewrite the function body starting from scratch.** The point of this exercise is to go through the process of finding the bugs and fixing the errors in existing code.
Hint: If it seems that some of your tests aren't running, check to make sure each test has a unique name that starts with `test`. If you have two tests with the same name, `unittest` won't complain but it will just run one of them!

(c) Switch driver and navigator for each function.

4. Write your own simple function with a bug inside of it, and put it in `buggy_functions.py`. Create a test file for this function, similar to the ones you made for the other ones. Make some tests that will pass and some that will fail.
Now, comment out the buggy lines and make the function work. Show your TA your buggy lines and test cases.

3 SQL Aggregate Functions

Download `lab10_sql.py` and `restaurants.db` from the Labs page. The function used to create the table is included for reference (you should not call it). Open the database in the SQLite browser on your computer if you can.
Task: Fill in the functions by running the proper queries. You will have to use some of the aggregate functions we've talked about in class (`SUM, AVG, MIN, MAX, COUNT`).

Congrats on completing the CSC120 Labs!!