Lab 2: Assertions, Exceptions, and Tesing

Ahmed Eldeeb

2023-08-21

Table of contents

Lab Exercise 1

Today we will spend some time getting familiar with some of the programming tools that can help make your code more robust and resilient to errors and boundary conditions. Tools like unit tests, exceptions, and asserts (and next week we will spend some time on debugging misbehaving code).

Testing is what you do when you finish implementing a piece of code and want to try it out to see if it works. Running your code manually and seeing if it works is a workable strategy for simple one-time scripts that do simple tasks, but there are situations (like writing a function that others will repeatedly use, or like running the same piece of code on hundreds of files or URLs) where it is prudent to test your code ahead of its actual use or deployment. Today we will use the pytest package to do that. We will also use some error handling techniques to make sure our code can handle malformed inputs.

Lab Exercise

1- Imagine a function that:

[option 1] takes in a string and return another string where the space separated words (or tokens) are the same as the input string but sorted according to a specific ordering. The ordering should be determined by the second argument to the function. The ordering could be specified as (1) lexicographic (alphabetic) according to the words (2) lexicographic according to the key produced by sorting the letters of the original word, or (3) numeric.

[option 2] (in case you want to practice regex and keep stick with the theme of this week's lecture) takes in a string and returns the first number in that string, returns `None` if there are no numeric values in the string.

2- Write an interface for that function (a function name and arguments), but do not implement the function yet (you can have it return an `None`, or an empty string for now). We will do this in a good old fashioned .py file (not a notebook or a quarto file).

3- Build a test suite using the pytest package to test that your function works as intended. Add at least 8 test cases with justification for each. Try to cover the main use cases, and as many potential corner cases or boundary conditions as possible.

4- Now run the test suite. It should fail for all your tests (unless one of them was passing an empty string).

5- Implement the function. You can do this at one go, or case by case. As you implement a case, you can rerun the test suite and see some of the tests relevant to that cases stopping to fail. When all the tests pass, you are done. This is called test-driven development.

6- If some cases are still failing, that's alright, we can use that failing code next week for demonstrating debugging functionality

7- Make sure you have assertions to check for invalid input types and combinations.

8- Make your function throw an exception for invalid input combinations.

9- Now write a loop that calls your function on a variety of inputs including invalid inputs.

10- In that loop, handle the thown exceptions and print something appropriate, but let the loop continue for later inputs.