C2M5L2_Unit_Tests_and_Edge_Cases-V3

May 1, 2022

1 Practice Notebook - Unit Tests and Edge Cases

Below we have some code that makes a list of specific letters found in any string. If you run it, you can see what it does.

```
In [8]: import re

        my_txt = "An investment in knowledge pays the best interest."

        def LetterCompiler(txt):
            result = re.findall(r'([a-c]).', txt)
            return result

        print(LetterCompiler(my_txt))
```

```
['a', 'b']
```

From the output, you can see that the `LetterCompiler( )` function finds all matches for the letters a through c in an input string if followed by another character and returns them as a list of strings, with each string representing one match. Nice. But can we be sure that this function will always do what we expect it to do? We need to write code to help us catch mistakes, errors and bugs. This code should automate the process of checking if the returned value of our code matches the expectations by dynamically feeding into it test cases. Since we're dynamically feeding in different strings, it would be prudent to create unit tests for our code. We can use the module **unittest** for this. Fill in the blanks below to create an automatic unint test that verifies whether input strings have the correct list of string matches.

```
In [2]: import unittest

        class TestCompiler(unittest.TestCase):

            def test_basic(self):
                testcase = "The best preparation for tomorrow is doing your best today."
                expected = ['b', 'a', 'a', 'b', 'a']
                self.assertEqual(LetterCompiler(testcase), expected)
```

Now that your automatic test is coded, you need to call the `unittest.main( )` function to run the test. It is important to note that the configuration for running unit tests in Jupyter is different than running unit tests from the command line. Running `unittest.main( )` in Jupyter **will result in an error**. You can see this by runnig the following cell to execute your automatic test.

```
In [3]: unittest.main()

E
======================================================================
ERROR: /home/jovyan/ (unittest.loader._FailedTest)
----------------------------------------------------------------------
AttributeError: module '__main__' has no attribute '/home/jovyan/'

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)



        An exception has occurred, use %tb to see the full traceback.


        SystemExit: True
```

Yikes! **SystemExit: True** means an error occurred, as expected. The reason is that `unittest.main( )` looks at *sys.argv*. In Jupyter, by default, the first parameter of *sys.argv* is what started the Jupyter kernel which is not the case when executing it from the command line. This default parameter is passed into `unittest.main( )` as an attribute when you don't explicitly pass it attributes and is therefore what causes the error about the kernel connection file not being a valid attribute. Passing an explicit list to `unittest.main( )` prevents it from looking at *sys.argv*. Let's pass it the list ['first-arg-is-ignored'] for example. In addition, we will pass it the parameter *exit* = *False* to prevent `unittest.main( )` from shutting down the kernel process. Run the following cell with the *argv* and *exit* parameters passed into `unittest.main( )` to rerun your automatic test.

```
In [4]: unittest.main(argv = ['first-arg-is-ignored'], exit = False)

.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK


Out[4]: <unittest.main.TestProgram at 0x7fbe26732b70>
```

Did your automatic test pass? Was **OK** the result? If not, go back to your automatic test code and make sure you filled in the blanks correctly. If your automatic test passed, great! You have successfully filled in the gaps to create an automatic test that verifies whether input strings have the correct list of string matches.

This is great work so far, but your automatic test includes only one test case. You need to make it grow. You can feed in more strings as test cases to test whether your code works in the general case. But you should also see what happens when you give it some input that you might not expect it to run into under normal operations. Edge cases are inputs to code that produce unexpected results, and are found at the extreme ends of the ranges of input we imagine programs will typically work with. Can you use the cell below to write some edge cases? We've already filled in another test case for you! As it is, this test will run fine. Can you come up with at least one test case that you think could result in a wrong return value? No wrong answers! Feel free to play around.

```
In [9]: class TestCompiler2(unittest.TestCase):

            def test_two(self):
                testcase = "A b c d e f g h i j k l m n o q r s t u v w x y z"
                expected = ['b', 'c']
                self.assertEqual(LetterCompiler(testcase), expected)

        # EDGE CASES HERE
            def test_empty(self):
                testcase = ""
                expected = ""
                self.assertEqual(LetterCompiler(testcase), expected)

        unittest.main(argv = ['first-arg-is-ignored'], exit = False)
```

```
.F.
======================================================================
FAIL: test_empty (__main__.TestCompiler2)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-9-9ae6adeaed70>", line 12, in test_empty
    self.assertEqual(LetterCompiler(testcase), expected)
AssertionError: [] != ''

----------------------------------------------------------------------
Ran 3 tests in 0.005s

FAILED (failures=1)


Out[9]: <unittest.main.TestProgram at 0x7fbe25e50fd0>
```

Did you find any edge cases? If not, continue working on it. Choosing test cases can be an exercise in creativity. Coming up with different ways a code might break can be super fun! When you have found an edge case, think about special handling in your script in order for your code

to continue to behave correctly. If you are out of ideas: Try removing the spaces and figure out why they were in the example testcase. Does that give you an idea for other tests? When you have found at least one edge case, you are all done with this notebook. You should take a moment to reflect on what you've done so far. It's super impressive and it's going to fit nicely in your IT toolkit.