Experiment No. 1

Title: Program on Unit Testing using Python

Batch: A3 **Roll No:16010421073** **Experiment No.:1**

Aim: Program on implementation of Unit Testing using Python

Resources needed: Python IDE

Theory:

What is Software Testing?

Software Testing involves execution of software/program components using manual to evaluate one or more properties of interest. The purpose of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements.

What is Unit Testing?

Unit testing is a technique in which particular module is tested to check by developer himself whether there are any errors. The primary focus of unit testing is test an individual unit of system to analyze, detect, and fix the errors.

What is the Python unittest?

Python provides the **unittest module** to test the unit of source code. The unittest plays an essential role when we are writing the huge code, and it provides the facility to check whether the output is correct or not.

Normally, we print the value and match it with the reference output or check the output manually.

Python testing framework uses Python's built-in **assert() function** which tests a particular condition. If the assertion fails, an AssertionError will be raised. The testing framework will then identify the test as Failure. Other exceptions are treated as Error.

The following three sets of assertion functions are defined in unittest module −

· Basic Boolean Asserts

· Comparative Asserts

· Asserts for Collections

Basic assert functions evaluate whether the result of an operation is True or False. All the assert methods accept a **msg** argument that, if specified, is used as the error message on failure.

You can write both integration tests and unit tests in Python. To write a unit test for the built-in function sum(), you would check the output of sum() against a known output.

For example, here's how you check that the sum() of the numbers (1, 2, 3) equals

6: >>> assert sum([1, 2, 3]) == 6, "Should be 6"

This will not output anything on the REPL because the values are correct.

If the result from sum() is incorrect, this will fail with an AssertionError and the message "Should be 6". Try an assertion statement again with the wrong values to see an AssertionError:


```
>>> assert sum([1, 1, 1]) == 6,
"Should be 6"

Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
AssertionError: Should be 6
```

In the REPL, you are seeing the raised AssertionError because the result of sum() does not match 6.

Instead of testing on the REPL, you'll want to put this into a new Python file called test_sum.py and execute it again:

```
def test_sum():
 assert sum([1, 2, 3]) == 6, "Should be 6"

if __name__ == "__main__":
 test_sum()
 print("Everything passed")
```

Now you have written a **test case**, an assertion, and an entry point (the command line). You can now execute this at the command line:


```
$ python test_sum.py
Everything passed
```

You can see the successful result, **Everything passed**.

In Python, sum() accepts any iterable as its first argument. You tested with a list. Now test with a tuple as well. Create a new file called test_sum_2.py with the following code:

```
def test_sum():
 assert sum([1, 2, 3]) == 6, "Should be 6"

def test_sum_tuple():
 assert sum((1, 2, 2)) == 6, "Should be 6"

if __name__ == "__main__":
 test_sum()
 test_sum_tuple()
 print("Everything passed")
```


When you execute test_sum_2.py, the script will give an error because the sum() of (1, 2, 2) is 5, not 6. The result of the script gives you the error message, the line of code, and the traceback:

```
$ python test_sum_2.py
Traceback (most recent call last):
 File "test_sum_2.py", line 9, in <module>
```

```
test_sum_tuple()
File "test_sum_2.py", line 5, in test_sum_tuple
assert sum((1, 2, 2)) == 6, "Should be 6"
AssertionError: Should be 6
```

Here you can see how a mistake in your code gives an error on the console with some information on where the error was and what the expected result was.

Writing tests in this way is okay for a simple check, but what if more than one fails? This is where test runners come in. The test runner is a special application designed for running tests,

checking the output, and giving you tools for debugging and diagnosing tests and

applications. Choosing a Test Runner

Python contains many test runners. The most popular build-in Python library is called **unittest.** The unittest is portable to the other frameworks. Consider the following three top most test runners.

- unittest
- nose Or nose2
- pytest

unittest

The unittest is built into the Python standard library since 2.1. The best thing about the unittest, it comes with both a test framework and a test runner. There are few requirements of the unittest to write and execute the code.

- The code must be written using the classes and functions.
- The sequence of distinct assertion methods in the **TestCase** class apart from the built-in asserts statements.

Let's implement the above example using the unittest case.

Example -

```
import unittest
class TestingSum(unittest.TestCase):
```

```
 def test_sum(self):
 self.assertEqual(sum([2, 3, 5]), 10, "It should be 10")
 def test_sum_tuple(self):
 self.assertEqual(sum((1, 3, 5)), 10, "It should be 10")

if __name__ == '__main__':
 unittest.main()
```

Output:


```
.F
-
FAIL: test_sum_tuple (__main__.TestingSum)
--
Traceback (most recent call last):
 File "<string>", line 11, in test_sum_tuple
AssertionError: 9 != 10 : It should be 10

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
Traceback (most recent call last):
 File "<string>", line 14, in <module>
 File "/usr/lib/python3.8/unittest/main.py", line 101, in __init__
 self.runTests()
 File "/usr/lib/python3.8/unittest/main.py", line 273, in runTests
 sys.exit(not
self.result.wasSuccessful())
 SystemExit: True
```

As we can see in the output, it shows the **dot(.)** for the successful execution and **F** for the one failure.

Activities:

Write a program to add multiple numbers in loop and test it by running minimum 3 test cases **Result: (script and output)**

Script:

mul file

```
def mul(arg):
    total=1
    for val in arg:
        total=total* val
 return total
```

Main File

```
import unittest
from my_mul import mul

class TestMul(unittest.TestCase):


    def test_int(self):
        data=[1,6,9]
        result=mul(data)
        self.assertEqual(result,96)

    def test_list_int(self):
        data=[2,3,4]
        result=mul(data)
        self.assertEqual(result,24)

    def test_mul_tuple(self):
        data=(4,5,6)
        result=mul(data)
        self.assertEqual(result,12)


if __name__ == '__main__':
    unittest.main()
```

OUTPUT:

```
F.F
======================================================================
FAIL: test_int (__main__.TestMul)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/exam/Desktop/.py1.py", line 10, in test_int
    self.assertEqual(result,76)
AssertionError: 54 != 76

======================================================================
FAIL: test_mul_tuple (__main__.TestMul)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/exam/Desktop/.py1.py", line 20, in test_mul_tuple
    self.assertEqual(result,30)
AssertionError: 120 != 30

----------------------------------------------------------------------
Ran 3 tests in 0.009s

FAILED (failures=2)
```

Outcomes:

Questions:

a) Differentiate between Unit Testing and Integration Testing.

Difference between Unit and Integration Testing:

Unit Testing	Integration Testing
• Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation.	• Integration testing -- also known as integration and testing (I&T) -- is a type of software testing in which the different units, modules or components of a software application are tested as a combined entity.
• Unit test is a white box testing.	• Integration testing is a black box testing.
• Less costly	• More costly
• Defects detection is easy	• Defects detection is hard.
• Unit testing are always performed before any other tests.	• Integration testing is conducted after unit testing and before system testing.
• Tests are performed by the developer.	• Tests are performed by the tester.

b) Differentiate between manual testing and automated testing.

Difference between Manual and Automated Testing:

Manual Testing	Automated Testing
• Test cases are executed manually.	• Test cases are executed with the help of tools.
• It is less costlier.	• It is more costlier.
• For some test cases it consumes time.	• As it is a machine it takes less time to execute cases.
• May not be applicable on different operating systems.	• You can easily test the application on different operating system.
• Difficult to execute all the test cases.	• You can achieve the test coverage target.
• Difficult to test an application on different browsers.	• You can test on different browsers

Conclusion: (Conclusion to bbased on the objectives and outcomes achieved)
In this experiment we learned about how to use unit testing with help of different test cases by using multiplation operator.

References:
1. Daniel Arbuckle, Learning Python Testing, Packt Publishing, 1st Edition, 2014 2. Wesly J Chun, Core Python Applications Programming, O'Reilly, 3rd Edition, 2015 3. Wes McKinney, Python for Data Analysis, O'Reilly, 1st Edition, 2017 4. Albert Lukaszewsk, MySQL for Python, Packt Publishing, 1st Edition, 2010 5. Eric Chou, Mastering Python Networking, Packt Publishing, 2nd Edition, 2017