Experiment No. 1

Title: Program on Unit Testing using Python

Batch:A2 **Roll No:16010421063** **Experiment 1**

No.:2 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 wh[illegible] is tested to check by developer himself whether there are any errors. The [illegible]esting is test an individual unit of system to analyze, detect, and fix the erro[illegible]

What is the Python unittest?

Python provides the **unittest mo[illegible]** [illegible] 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,


Traceback (most recent call las
  File "<stdin>", line 1, in <mo
 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")
```

V _sum_2.py, the script will give an error because the sum() of (1, 2, 2) is 5, n script gives you the error message, the line of code, and the traceback:

```
py
nt call last):
", line 9, in <module>

", line 5, in test_sum_tuple
)) == 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:

```
  File "<string>", line 11, in
test_sum_tuple AssertionError: 9 != 10 :
-----------------------------------------------
It should be 10
```

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

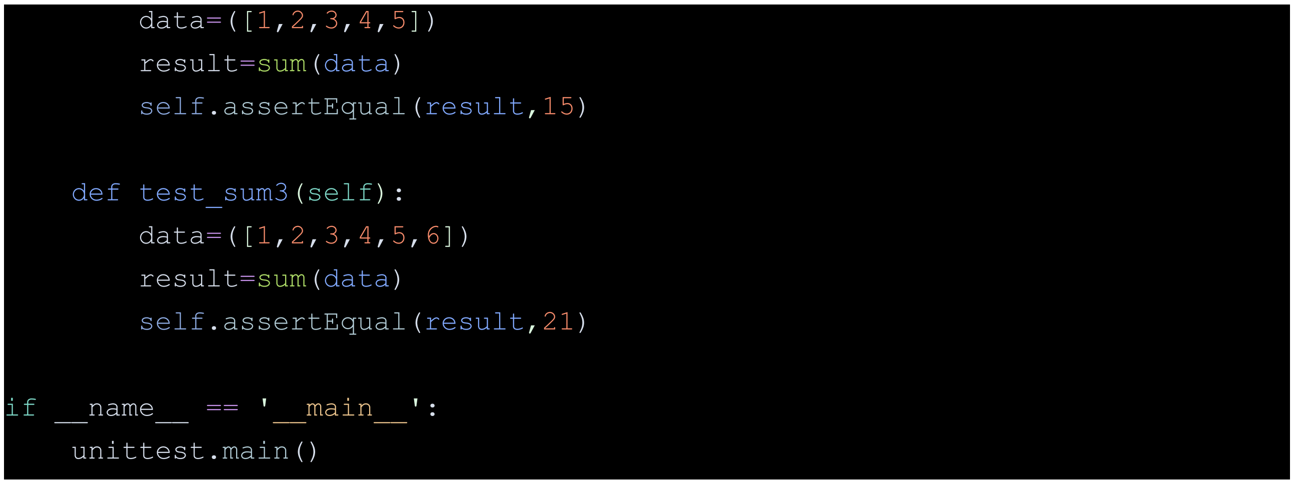
Activities:

Write a program to add multiple numbers in loop and test it by running minimum 3 test cases

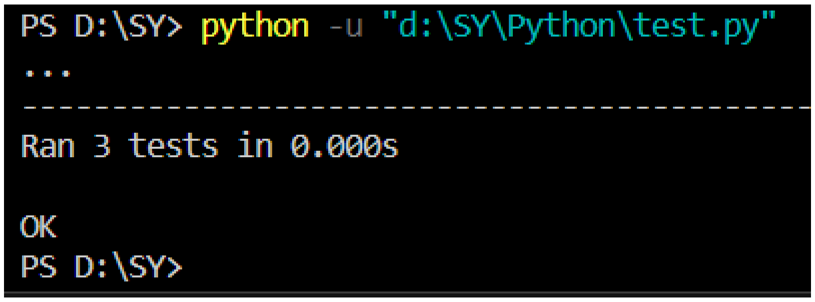
---

Result: (script and output)

```
def sum(args):
    total = 0
    for arg in args:
        total += arg
    return total

import unittest
class TestSum(unittest.TestCase):
    def test_sum(self):
        data = [1, 2, 3]
        result = sum(data)
        self.assertEqual(result, 6)

    def test_sum2(self):
```



```
        data=([1,2,3,4,5])
        result=sum(data)
        self.assertEqual(result,15)

    def test_sum3(self):
        data=([1,2,3,4,5,6])
        result=sum(data)
        self.assertEqual(result,21)

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



```
PS D:\SY> python -u "d:\SY\Python\test.py"
...
----------------------------------------------
Ran 3 tests in 0.000s

OK
PS D:\SY>
```

Outcomes:

This experiment is of Module 0

Questions:

a) Differentiate between Unit Testing and Integration Testing.

In unit testing, each module of the software is tested separately. In integration testing, all modules of the software are tested combined. In unit testing tester knows the internal design of the software. Integration testing doesn't know the internal design of the software.Detection of defects in unit testing is easy. Detection of defects in integration testing is difficult.

b) Differentiate between manual testing and automated testing.

Automation Testing executes test cases with the help of automation technologies. in Manual Testing, the human tester and software perform test cases in manual testing. Automated testing is much quicker than manual testing. Manual testing takes time and requires human resources. Automated testing does not allow for exploratory testing. Manual Testing allows for exploratory testing. When doing the same set of test cases regularly, automation testing is a good option. When a test case simply has to run once or twice, manual testing comes in handy.

Conclusion: (Conclusion to be based on the objectives and outcomes achieved)
We learnt about the concept of testing our code by using libraries such as "unittest". This allowed us to write testcases and verify whether our code is working properly

---

[illegible]

[illegible] e, Learning Python Testing, Packt Publishing, 1st Edition, 2014

[illegible] Core Python Applications Programming, O'Reilly, 3rd Edition, 2015

[illegible], Python for Data Analysis, O'Reilly, 1st Edition, 2017

[illegible] wsk, MySQL for Python, Packt Publishing, 1st Edition, 2010

[illegible] tering Python Networking, Packt Publishing, 2nd Edition, 2017