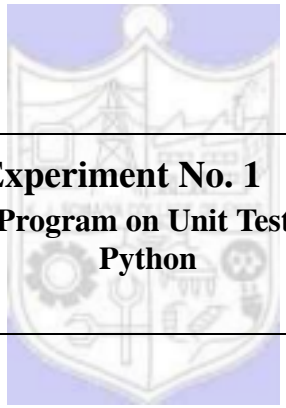

Experiment No. 1

Title: Program on Unit Testing using Python

Name: Chandana Ramesh Galgali **Batch: B-1** **Roll No: 16010422234**

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 manuals 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 a particular module is tested to check by the developer himself whether there are any errors. The primary focus of unit testing is to test an individual unit of the 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 built-in Python library is called **unittest.** The unittest is portable to the other frameworks. Consider the following three top 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 a loop and test it by running minimum 3 test cases.

Result: (script and output)

```python
import unittest


def add_numbers(numbers):
    total = 0
    for num in numbers:
        total += num
    return total


class TestAddNumbers(unittest.TestCase):
    def test_add_numbers(self):
        self.assertEqual(add_numbers([1, 2, 3]), 6, "It should be 6")
    def test_add_numbers(self):
        self.assertEqual(sum([2, 3, 5]), 10, "It should be 10")
    def test_add_numbers(self):
        self.assertEqual(sum((1, 3, 5)), 10, "It should be 10")
```

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

Outcomes:

```
F
======================================================================
FAIL: test_add_numbers (__main__.TestAddNumbers.test_add_numbers)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "c:\Users\daxay\PL-py\EXP-1\Untitled-1.py", line 15, in test_add_numbers
    self.assertEqual(sum((1, 3, 5)), 10, "It should be 10")
AssertionError: 9 != 10 : It should be 10

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

FAILED (failures=1)
```

Questions:

a) Differentiate between Unit Testing and Integration Testing.

Unit testing and integration testing are two different types of software testing that serve distinct purposes in the software development process.

Unit Testing:

- Focuses on testing individual units or components of code in isolation.
- Typically performed by developers during the development phase.
- Aims to verify the correctness of each unit's functionality and behavior.
- Uses test cases to validate the unit's inputs, outputs, and internal logic.
- Mock objects or stubs may be used to simulate dependencies and isolate the unit being tested.
- Helps identify and fix defects early in the development cycle.
- Provides a foundation for building reliable and maintainable code.

Integration Testing:

- Focuses on testing the interaction between different modules or components of a software
- system.
- Performed after unit testing and before system testing.
- Verifies that the integrated components work together as expected.
- Tests the interfaces, data flow, and communication between modules.
- Ensures that the integrated system functions correctly and meets the specified requirements.
- May involve both top-down and bottom-up approaches to integration.
- Helps identify issues related to module integration, such as data inconsistencies or communication failures.

In summary, unit testing primarily focuses on testing individual units of code, while integration testing focuses on testing the interaction and integration between different components or modules. Both types of testing are essential for ensuring the quality and reliability of a software system.

b) Differentiate between manual testing and automated testing.

Manual testing and automated testing are two approaches to software testing, each with its own advantages and considerations.

Manual Testing:

- Involves human testers executing test cases and validating the software's behavior.
- Requires manual intervention and observation throughout the testing process.
- Provides flexibility to explore and adapt test scenarios based on tester's intuition and experience.
- Well-suited for exploratory testing, usability testing, and ad-hoc testing.
- Requires significant time and effort, especially for repetitive or complex test cases.
- Prone to human errors and inconsistencies.
- Ideal for small-scale projects or when the software is constantly changing.

Automated Testing:

- Involves using software tools and scripts to execute test cases and compare actual results with expected results.
- Offers repeatability and consistency in test execution.
- Allows for faster and more efficient testing, especially for large-scale projects or regression testing.
- Reduces the risk of human errors and increases test coverage.
- Requires upfront investment in test automation frameworks and scripts.
- May not be suitable for all types of testing, such as usability testing or exploratory testing.
- Requires maintenance and updates as the software evolves.

In summary, manual testing relies on human testers to execute test cases, providing flexibility and adaptability but requiring more time and effort. Automated testing, on the other hand, utilizes software tools and scripts to execute tests, offering repeatability and efficiency but requiring upfront investment and ongoing maintenance. The choice between manual and automated testing depends on factors such as project size, complexity, test requirements, and available resources. Often, a combination of both approaches is used to achieve comprehensive test coverage.

Conclusion: (Conclusion to be based on the objectives and outcomes achieved)

We could gain a comprehensive understanding of unit testing principles and techniques in Python. We were able to apply our knowledge to write effective unit tests, ensuring the quality and reliability of our code. The program fostered a culture of test-driven development, promoting better code organization and maintainability.

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