UnitTest Framework - Framework

'unittest' supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.

The unittest module provides classes that make it easy to support these qualities for a set of tests.

To achieve this, unittest supports the following important concepts −

- **test fixture** − This represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.
- **test case** − This is the smallest unit of testing. This checks for a specific response to a particular set of inputs. unittest provides a base class, **TestCase**, which may be used to create new test cases.
- **test suite** − This is a collection of test cases, test suites, or both. This is used to aggregate tests that should be executed together. Test suites are implemented by the TestSuite class.
- **test runner** − This is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

Creating a Unit Test

The following steps are involved in writing a simple unit test −

Step 1 − Import the unittest module in your program.

Step 2 − Define a function to be tested. In the following example, add() function is to be subjected to test.

Step 3 − Create a testcase by subclassing unittest.TestCase.

Step 4 − Define a test as a method inside the class. Name of method must start with 'test'.

Step 5 − Each test calls assert function of TestCase class. There are many types of asserts. Following example calls assertEquals() function.

Step 6 − assertEquals() function compares result of add() function with arg2 argument and throws assertionError if comparison fails.

Step 7 − Finally, call main() method from the unittest module.

Live Demo

```
import unittest
def add(x,y):
   return x + y

class SimpleTest(unittest.TestCase):
   def testadd1(self):
      self.assertEquals(add(4,5),9)

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

Step 8 − Run the above script from the command line.

```
C:\Python27>python SimpleTest.py
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Step 9 − The following three could be the possible outcomes of a test −

Sr.No	Message & Description
1	**OK** The test passes. ‘A’ is displayed on console.
2	**FAIL** The test does not pass, and raises an AssertionError exception. ‘F’ is displayed on console.
3	**ERROR** The test raises an exception other than AssertionError. ‘E’ is displayed on console.

These outcomes are displayed on the console by '.', 'F' and 'E' respectively.

Command Line Interface

The unittest module can be used from the command line to run single or multiple tests.

```
python -m unittest test1
python -m unittest test_module.TestClass
python -m unittest test_module.TestClass.test_method
```

unittest supports the following command line options. For a list of all the command-line options, use the following command −

```
Python –m unittest -h
```

Sr.No	Option & Description
1	**-h, --help** Show this message
2	**v, --verbose** Verbose output
3	**-q, --quiet** Minimal output
4	**-f, --failfast** Stop on first failure
5	**-c, --catch** Catch control-C and display results
6	**-b, --buffer** Buffer stdout and stderr during test runs