UnitTest Framework - Quick Guide

UnitTest Framework - Overview

Unit testing is a software testing method by which individual units of source code, such as functions, methods, and class are tested to determine whether they are fit for use. Intuitively, one can view a unit as the smallest testable part of an application. Unit tests are short code fragments created by programmers during the development process. It forms the basis for component testing.

Unit testing can be done in the following two ways −

Manual Testing	Automated Testing
Executing the test cases manually without any tool support is known as manual testing. • Since test cases are executed by human resources so it is very **time consuming and tedious**. • As test cases need to be executed manually so more testers are required in manual testing. • It is less reliable as tests may not be performed with precision each time because of human errors. • No programming can be done to write sophisticated tests which fetch hidden information.	Taking tool support and executing the test cases by using automation tool is known as automation testing. • Fast Automation runs test cases significantly faster than human resources. • The **investment over human resources is less** as test cases are executed by using automation tool. • Automation tests perform precisely same operation each time they are run and **are more reliable**. • Testers **can program sophisticated tests** to bring out hidden information.

JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit that originated with JUnit. You can find out JUnit Tutorial here.

The Python unit testing framework, sometimes referred to as “PyUnit,” is a Python language version of JUnit developed by Kent Beck and Erich Gamma. PyUnit forms part of the Python Stand

Library as of Python version 2.1.

Python unit testing framework 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.

This tutorial has been prepared for the beginners to help them understand the basic functionality of Python testing framework. After completing this tutorial you will find yourself at a moderate level of expertise in using Python testing framework from where you can take yourself to the next levels.

You should have reasonable expertise in software development using Python language. Our Python tutorial is a good place to start learning Python. Knowledge of basics of software testing is also desirable.

Environment Setup

The classes needed to write tests are to be found in the 'unittest' module. If you are using older versions of Python (prior to Python 2.1), the module can be downloaded from http://pyunit.sourceforge.net/ . However, unittest module is now a part of the standard Python distribution; hence it requires no separate installation.

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 retur

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.

```
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

UnitTest Framework - API

This chapter discusses the classes and methods defined in the unittest module. There are five major classes in this module.

TestCase Class

Object of this class represents the smallest testable unit. It holds the test routines and provides hooks for preparing each routine and for cleaning up thereafter.

The following methods are defined in the TestCase class −

Sr.No	Method & Description
1	**setUp()** Method called to prepare the test fixture. This is called immediately before calling the test method
2	**tearDown()** Method called immediately after the test method has been called and the result recorded. This is called even if the test method raised an exception,
3	**setUpClass()** A class method called before tests in an individual class run.
4	**tearDownClass()** A class method called after tests in an individual class have run.
5	**run(result = None)** Run the test, collecting the result into the test result object passed as *result*.
6	**skipTest(reason)** Calling this during a test method or setUp() skips the current test.
7	**debug()** Run the test without collecting the result.
8	**shortDescription()** Returns a one-line description of the test.

Fixtures

There can be numerous tests written inside a TestCase class. These test methods may need database connection, temporary files or other resources to be initialized. These are called fixtu

TestCase includes a special hook to configure and clean up any fixtures needed by your tests. To configure the fixtures, override setUp(). To clean up, override tearDown().

In the following example, two tests are written inside the TestCase class. They test result of addition and subtraction of two values. The setup() method initializes the arguments based on shortDescription() of each test. teardown() method will be executed at the end of each test.

```
import unittest

class simpleTest2(unittest.TestCase):
   def setUp(self):
      self.a = 10
      self.b = 20
      name = self.shortDescription()
      if name == "Add":
         self.a = 10
         self.b = 20
         print name, self.a, self.b
      if name == "sub":
         self.a = 50
         self.b = 60
         print name, self.a, self.b
   def tearDown(self):
      print '\nend of test',self.shortDescription()

   def testadd(self):
      """Add"""
      result = self.a+self.b
      self.assertTrue(result == 100)
   def testsub(self):
      """sub"""
      result = self.a-self.b
      self.assertTrue(result == -10)

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

Run the above code from the command line. It gives the following output −

```
C:\Python27>python test2.py
Add 10 20
F
end of test Add
```

```
sub 50 60
end of test sub
.
======================================================================
FAIL: testadd (__main__.simpleTest2)
Add
----------------------------------------------------------------------
Traceback (most recent call last):
   File "test2.py", line 21, in testadd
      self.assertTrue(result == 100)
AssertionError: False is not true
----------------------------------------------------------------------
Ran 2 tests in 0.015s

FAILED (failures = 1)
```

Class Fixture

TestCase class has a setUpClass() method which can be overridden to execute before the execution of individual tests inside a TestCase class. Similarly, tearDownClass() method will be executed after all test in the class. Both the methods are class methods. Hence, they must be decorated with @classmethod directive.

The following example demonstrates the use of these class methods −

```
import unittest

class TestFixtures(unittest.TestCase):

   @classmethod
   def setUpClass(cls):
      print 'called once before any tests in class'

   @classmethod
   def tearDownClass(cls):
      print '\ncalled once after all tests in class'

   def setUp(self):
      self.a = 10
      self.b = 20
      name = self.shortDescription()
      print '\n',name
   def tearDown(self):
```

```
      print '\nend of test',self.shortDescription()

   def test1(self):
      """One"""
      result = self.a+self.b
      self.assertTrue(True)
   def test2(self):
      """Two"""
      result = self.a-self.b
      self.assertTrue(False)

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

TestSuite Class

Python's testing framework provides a useful mechanism by which test case instances can be grouped together according to the features they test. This mechanism is made available by TestSuite class in unittest module.

The following steps are involved in creating and running a test suite.

Step 1 − Create an instance of TestSuite class.

```
suite = unittest.TestSuite()
```

Step 2 − Add tests inside a TestCase class in the suite.

```
suite.addTest(testcase class)
```

Step 3 − You can also use makeSuite() method to add tests from a class

```
suite = unittest.makeSuite(test case class)
```

Step 4 − Individual tests can also be added in the suite.

```
suite.addTest(testcaseclass(""testmethod")
```

Step 5 − Create an object of the TestTestRunner class.

```
runner = unittest.TextTestRunner()
```

Step 6 − Call the run() method to run all the tests in the suite

```
runner.run (suite)
```

The following methods are defined in TestSuite class −

Sr.No	Method & Description
1	**addTest()** Adds a test method in the test suite.
2	**addTests()** Adds tests from multiple TestCase classes.
3	**run()** Runs the tests associated with this suite, collecting the result into the test result object
4	**debug()** Runs the tests associated with this suite without collecting the result.
5	**countTestCases()** Returns the number of tests represented by this test object

The following example shows how to use TestSuite class −

```
import unittest
class suiteTest(unittest.TestCase):
   def setUp(self):
      self.a = 10
      self.b = 20

   def testadd(self):
      """Add"""
```

```
      result = self.a+self.b
      self.assertTrue(result == 100)
   def testsub(self):
      """sub"""
      result = self.a-self.b
      self.assertTrue(result == -10)

def suite():
   suite = unittest.TestSuite()
##   suite.addTest (simpleTest3("testadd"))
##   suite.addTest (simpleTest3("testsub"))
   suite.addTest(unittest.makeSuite(simpleTest3))
   return suite

if __name__ == '__main__':
   runner = unittest.TextTestRunner()
   test_suite = suite()
   runner.run (test_suite)
```

You can experiment with the addTest() method by uncommenting the lines and comment statement having makeSuite() method.

TestLoader Class

The unittest package has the TestLoader class which is used to create test suites from classes and modules. By default, the unittest.defaultTestLoader instance is automatically created when the unittest.main(0 method is called. An explicit instance, however enables the customization of certain properties.

In the following code, tests from two classes are collected in a List by using the TestLoader object.

```
import unittest
testList = [Test1, Test2]
testLoad = unittest.TestLoader()

TestList = []
for testCase in testList:
   testSuite = testLoad.loadTestsFromTestCase(testCase)
   TestList.append(testSuite)

newSuite = unittest.TestSuite(TestList)
```

```
runner = unittest.TextTestRunner()
runner.run(newSuite)
```

The following table shows a list of methods in the TestLoader class −

S.No	Method & Description
1	**loadTestsFromTestCase()** Return a suite of all tests cases contained in a TestCase class
2	**loadTestsFromModule()** Return a suite of all tests cases contained in the given module.
3	**loadTestsFromName()** Return a suite of all tests cases given a string specifier.
4	**discover()** Find all the test modules by recursing into subdirectories from the specified start directory, and return a TestSuite object

TestResult Class

This class is used to compile information about the tests that have been successful and the tests that have met failure. A TestResult object stores the results of a set of tests. A TestResult instance is returned by the TestRunner.run() method.

TestResult instances have the following attributes −

Sr.No	Attribute & Description
1	**Errors** A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents a test which raised an unexpected exception.
2	**Failures** A list containing 2-tuples of TestCase instances and strings holding formatted tracebacks. Each tuple represents a test where a failure was explicitly signalled using the TestCase.assert*() methods.
3	**Skipped** A list containing 2-tuples of TestCase instances and strings holding the reason for skipping the test.
4	**wasSuccessful()** Return True if all tests run so far have passed, otherwise returns False.
5	**stop()** This method can be called to signal that the set of tests being run should be aborted.
6	**startTestRun()** Called once before any tests are executed.
7	**stopTestRun()** Called once after all tests are executed.
8	**testsRun** The total number of tests run so far.
9	**Buffer**

	If set to true, **sys.stdout** and **sys.stderr** will be buffered in between startTest() and stopTest() being called.

The following code executes a test suite −

```
if __name__ == '__main__':
   runner = unittest.TextTestRunner()
   test_suite = suite()
   result = runner.run (test_suite)

   print "---- START OF TEST RESULTS"
   print result

   print "result::errors"
   print result.errors

   print "result::failures"
   print result.failures

   print "result::skipped"
   print result.skipped

   print "result::successful"
   print result.wasSuccessful()

   print "result::test-run"
   print result.testsRun
   print "---- END OF TEST RESULTS"
```

The code when executed displays the following output −

```
---- START OF TEST RESULTS
<unittest.runner.TextTestResult run = 2 errors = 0 failures = 1>
result::errors
[]
result::failures
[(<__main__.suiteTest testMethod = testadd>, 'Traceback (most recent call last):\n
   File "test3.py", line 10, in testadd\n
   self.assertTrue(result == 100)\nAssert
   ionError: False is not true\n')]
result::skipped
[]
```

```
result::successful
False
result::test-run
2
---- END OF TEST RESULTS
```

UnitTest Framework - Assertion

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.

Sr.No	Method & Description
1	**assertEqual(arg1, arg2, msg = None)** Test that *arg1* and *arg2* are equal. If the values do not compare equal, the test will fail.
2	**assertNotEqual(arg1, arg2, msg = None)** Test that *arg1* and *arg2* are not equal. If the values do compare equal, the test will fail.
3	**assertTrue(expr, msg = None)** Test that *expr* is true. If false, test fails
4	**assertFalse(expr, msg = None)** Test that *expr* is false. If true, test fails
5	**assertIs(arg1, arg2, msg = None)** Test that *arg1* and *arg2* evaluate to the same object.
6	**assertIsNot(arg1, arg2, msg = None)** Test that *arg1* and *arg2* don’t evaluate to the same object.
7	**assertIsNone(expr, msg = None)** Test that *expr* is None. If not None, test fails
8	**assertIsNotNone(expr, msg = None)** Test that *expr* is not None. If None, test fails
9	**assertIn(arg1, arg2, msg = None)** Test that *arg1* is in *arg2*.
10	**assertNotIn(arg1, arg2, msg = None)** Test that *arg1* is not in *arg2*.

11	**assertIsInstance(obj, cls, msg = None)** Test that *obj* is an instance of *cls*
12	**assertNotIsInstance(obj, cls, msg = None)** Test that *obj* is not an instance of *cls*

Some of the above assertion functions are implemented in the following code −

```
import unittest

class SimpleTest(unittest.TestCase):
   def test1(self):
      self.assertEqual(4 + 5,9)
   def test2(self):
      self.assertNotEqual(5 * 2,10)
   def test3(self):
      self.assertTrue(4 + 5 == 9,"The result is False")
   def test4(self):
      self.assertTrue(4 + 5 == 10,"assertion fails")
   def test5(self):
      self.assertIn(3,[1,2,3])
   def test6(self):
      self.assertNotIn(3, range(5))

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

When the above script is run, test2, test4 and test6 will show failure and others run successfully.

```
FAIL: test2 (__main__.SimpleTest)
----------------------------------------------------------------------
Traceback (most recent call last):
   File "C:\Python27\SimpleTest.py", line 9, in test2
      self.assertNotEqual(5*2,10)
AssertionError: 10 == 10

FAIL: test4 (__main__.SimpleTest)
----------------------------------------------------------------------
Traceback (most recent call last):
```

```
   File "C:\Python27\SimpleTest.py", line 13, in test4
      self.assertTrue(4+5==10,"assertion fails")
AssertionError: assertion fails

FAIL: test6 (__main__.SimpleTest)
----------------------------------------------------------------------
Traceback (most recent call last):
   File "C:\Python27\SimpleTest.py", line 17, in test6
      self.assertNotIn(3, range(5))
AssertionError: 3 unexpectedly found in [0, 1, 2, 3, 4]

----------------------------------------------------------------------
Ran 6 tests in 0.001s

FAILED (failures = 3)
```

The second set of assertion functions are **comparative asserts −**

- **assertAlmostEqual** (first, second, places = 7, msg = None, delta = None)

  Test that *first* and *second* are approximately (or not approximately) equal by computing the difference, rounding to the given number of decimal *places* (default 7),

- **assertNotAlmostEqual** (first, second, places, msg, delta)

  Test that first and second are not approximately equal by computing the difference, rounding to the given number of decimal places (default 7), and comparing to zero.

  In both the above functions, if delta is supplied instead of places then the difference between first and second must be less or equal to (or greater than) delta.

  Supplying both delta and places raises a TypeError.

- **assertGreater** (first, second, msg = None)

  Test that *first* is greater than *second* depending on the method name. If not, the test will fail.

- **assertGreaterEqual** (first, second, msg = None)

  Test that *first* is greater than or equal to *second* depending on the method name. If not, the test will fail

- **assertLess** (first, second, msg = None)

  Test that *first* is less than *second* depending on the method name. If not, the test will fail

- **assertLessEqual** (first, second, msg = None)

Test that *first* is less than or equal to *second* depending upon the method name. If not, the test will fail.

- **assertRegexpMatches** (text, regexp, msg = None)

  Test that a regexp search matches the text. In case of failure, the error message will include the pattern and the text. regexp may be a regular expression object or a string containing a regular expression suitable for use by **re.search()**.

- **assertNotRegexpMatches** (text, regexp, msg = None)

  Verifies that a *regexp* search does not match *text*. Fails with an error message including the pattern and the part of *text* that matches. *regexp* may be a regular expression object or a string containing a regular expression suitable for use by **re.search()**.

The assertion functions are implemented in the following example −

```
import unittest
import math
import re

class SimpleTest(unittest.TestCase):
   def test1(self):
      self.assertAlmostEqual(22.0/7,3.14)
   def test2(self):
      self.assertNotAlmostEqual(10.0/3,3)
   def test3(self):
      self.assertGreater(math.pi,3)
   def test4(self):
      self.assertNotRegexpMatches("Tutorials Point (I) Private Limited","Point")

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

The above script reports test1 and test4 as Failure. In test1, the division of 22/7 is not within 7 decimal places of 3.14. Similarly, since the second argument matches with the text in first argument, test4 results in AssertionError.

```
======================================================FAIL: test1 (__main__.SimpleTest)
----------------------------------------------------------------------
Traceback (most recent call last):
   File "asserttest.py", line 7, in test1
      self.assertAlmostEqual(22.0/7,3.14)
AssertionError: 3.142857142857143 != 3.14 within 7 places
```

```
======================================================================
FAIL: test4 (__main__.SimpleTest)
----------------------------------------------------------------------
Traceback (most recent call last):
   File "asserttest.py", line 13, in test4
      self.assertNotRegexpMatches("Tutorials Point (I) Private Limited","Point")
AssertionError: Regexp matched: 'Point' matches 'Point' in 'Tutorials Point (I)
Private Limited'
----------------------------------------------------------------------

Ran 4 tests in 0.001s

FAILED (failures = 2)
```

Assert for Collections

This set of assert functions are meant to be used with collection data types in Python, such as List, Tuple, Dictionary and Set.

Sr.No	Method & Description
1	**assertListEqual (list1, list2, msg = None)** Tests that two lists are equal. If not, an error message is constructed that shows only the differences between the two.
2	**assertTupleEqual (tuple1, tuple2, msg = None)** Tests that two tuples are equal. If not, an error message is constructed that shows only the differences between the two.
3	**assertSetEqual (set1, set2, msg = None)** Tests that two sets are equal. If not, an error message is constructed that lists the differences between the sets.
4	**assertDictEqual (expected, actual, msg = None)** Test that two dictionaries are equal. If not, an error message is constructed that shows the differences in the dictionaries.

The following example implements the above methods −

```
import unittest

class SimpleTest(unittest.TestCase):
   def test1(self):
      self.assertListEqual([2,3,4], [1,2,3,4,5])
   def test2(self):
      self.assertTupleEqual((1*2,2*2,3*2), (2,4,6))
   def test3(self):
      self.assertDictEqual({1:11,2:22},{3:33,2:22,1:11})

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

In the above example, test1 and test3 show AssertionError. Error message displays the differences in List and Dictionary objects.

```
FAIL: test1 (__main__.SimpleTest)
----------------------------------------------------------------------
Traceback (most recent call last):
   File "asserttest.py", line 5, in test1
      self.assertListEqual([2,3,4], [1,2,3,4,5])
AssertionError: Lists differ: [2, 3, 4] != [1, 2, 3, 4, 5]

First differing element 0:
2
1

Second list contains 2 additional elements.
First extra element 3:
4

- [2, 3, 4]
+ [1, 2, 3, 4, 5]
? +++       +++

FAIL: test3 (__main__.SimpleTest)
----------------------------------------------------------------------
Traceback (most recent call last):
   File "asserttest.py", line 9, in test3
      self.assertDictEqual({1:11,2:22},{3:33,2:22,1:11})
AssertionError: {1: 11, 2: 22} != {1: 11, 2: 22, 3: 33}
- {1: 11, 2: 22}
```

```
+ {1: 11, 2: 22, 3: 33}
?              +++++++

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures = 2)
```

UnitTest Framework - Test Discovery

The TestLoader class has a discover() function. Python testing framework uses this for simple test discovery. In order to be compatible, modules and packages containing tests must be importable from top level directory.

The following is the basic command line usage of test discovery −

```
Python –m unittest discover
```

Interpreter tries to load all modules containing test from current directory and inner directories recursively. Other command line options are −

Sr.No	Options & Description
1	**-v, --verbose** Verbose output
2	**-s, --start-directory** directory Directory to start discovery (. default)
3	**-p, --pattern** pattern Pattern to match test files (test*.py default)
4	**-t, --top-level-directory** directory Top level directory of project (defaults to start directory)

For example, in order to discover the tests in modules whose names start with 'assert' in 'tests' directory, the following command line is used −

```
C:\python27>python –m unittest –v –s "c:\test" –p "assert*.py"
```

Test discovery loads tests by importing them. Once test discovery has found all the test files from the start directory you specify, it turns the paths into package names to import.

If you supply the start directory as a package name rather than a path to a directory then discover assumes that whichever location it imports from is the location you intended, so you will not get the warning.

UnitTest Framework - Skip Test

Support for skipping tests has been added since Python 2.7. It is possible to skip individual test method or TestCase class, conditionally as well as unconditionally. The framework allows a certain test to be marked as an 'expected failure'. This test will 'fail' but will not be counted as failed in TestResult.

To skip a method unconditionally, the following unittest.skip() class method can be used −

```
import unittest

   def add(x,y):
      return x+y

class SimpleTest(unittest.TestCase):
   @unittest.skip("demonstrating skipping")
   def testadd1(self):
      self.assertEquals(add(4,5),9)

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

Since skip() is a class method, it is prefixed by @ token. The method takes one argument: a log message describing the reason for the skip.

When the above script is executed, the following result is displayed on console −

```
C:\Python27>python skiptest.py
s
----------------------------------------------------------------------
```

```
Ran 1 test in 0.000s

OK (skipped = 1)
```

The character 's' indicates that a test has been skipped.

Alternate syntax for skipping test is using instance method skipTest() inside the test function.

```
def testadd2(self):
   self.skipTest("another method for skipping")
   self.assertTrue(add(4 + 5) == 10)
```

The following decorators implement test skipping and expected failures −

Sr.No	Method & Description
1	**unittest.skip(reason)** Unconditionally skip the decorated test. *reason* should describe why the test is being skipped.
2	**unittest.skipIf(condition, reason)** Skip the decorated test if condition is true.
3	**unittest.skipUnless(condition, reason)** Skip the decorated test unless condition is true.
4	**unittest.expectedFailure()** Mark the test as an expected failure. If the test fails when run, the test is not counted as a failure.

The following example demonstrates the use of conditional skipping and expected failure.

```
import unittest

class suiteTest(unittest.TestCase):
   a = 50
```

```
      b = 40

      def testadd(self):
         """Add"""
         result = self.a+self.b
         self.assertEqual(result,100)

      @unittest.skipIf(a>b, "Skip over this routine")
      def testsub(self):
         """sub"""
         result = self.a-self.b
         self.assertTrue(result == -10)

      @unittest.skipUnless(b == 0, "Skip over this routine")
      def testdiv(self):
         """div"""
         result = self.a/self.b
         self.assertTrue(result == 1)

      @unittest.expectedFailure
      def testmul(self):
         """mul"""
         result = self.a*self.b
         self.assertEqual(result == 0)

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

In the above example, testsub() and testdiv() will be skipped. In the first case a>b is true, while in the second case b == 0 is not true. On the other hand, testmul() has been marked as expected failure.

When the above script is run, two skipped tests show 's' and the expected failure is shown as 'x'.

```
C:\Python27>python skiptest.py
Fsxs
======================================================================
FAIL: testadd (__main__.suiteTest)
Add
----------------------------------------------------------------------
Traceback (most recent call last):
   File "skiptest.py", line 9, in testadd
```

```
      self.assertEqual(result,100)
AssertionError: 90 != 100

----------------------------------------------------------------------
Ran 4 tests in 0.000s

FAILED (failures = 1, skipped = 2, expected failures = 1)
```

UnitTest Framework - Exceptions Test

Python testing framework provides the following assertion methods to check that exceptions are raised.

assertRaises(exception, callable, *args, **kwds)

Test that an exception (first argument) is raised when a function is called with any positional or keyword arguments. The test passes if the expected exception is raised, is an error if another exception is raised, or fails if no exception is raised. To catch any of a group of exceptions, a tuple containing the exception classes may be passed as exception.

In the example below, a test function is defined to check whether ZeroDivisionError is raised.

```
import unittest

def div(a,b):
   return a/b
class raiseTest(unittest.TestCase):
   def testraise(self):
      self.assertRaises(ZeroDivisionError, div, 1,0)

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

The testraise() function uses assertRaises() function to see if division by zero occurs when div() function is called. The above code will raise an exception. But changes arguments to div() function as follows −

```
self.assertRaises(ZeroDivisionError, div, 1,1)
```

When a code is run with these changes, the test fails as ZeroDivisionError doesn't occur.

```
F
======================================================================
FAIL: testraise (__main__.raiseTest)
----------------------------------------------------------------------
Traceback (most recent call last):
   File "raisetest.py", line 7, in testraise
      self.assertRaises(ZeroDivisionError, div, 1,1)
AssertionError: ZeroDivisionError not raised

----------------------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures = 1)
```

assertRaisesRegexp(exception, regexp, callable, *args, **kwds)

Tests that *regexp* matches on the string representation of the raised exception. regexp may be a regular expression object or a string containing a regular expression suitable for use by re.search().

The following example shows how assertRaisesRegexp() is used −

```
import unittest
import re

class raiseTest(unittest.TestCase):
   def testraiseRegex(self):
      self.assertRaisesRegexp(TypeError, "invalid", reg,"Point","TutorialsPoint")

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

Here, testraseRegex() test doesn't fail as first argument. "Point" is found in the second argument string.

```
======================================================================
FAIL: testraiseRegex (__main__.raiseTest)
----------------------------------------------------------------------
Traceback (most recent call last):
   File "C:/Python27/raiseTest.py", line 11, in testraiseRegex
      self.assertRaisesRegexp(TypeError, "invalid", reg,"Point","TutorialsPoint")
AssertionError: TypeError not raised
----------------------------------------------------------------------
```

However, the change is as shown below −

```
self.assertRaisesRegexp(TypeError, "invalid", reg,123,"TutorialsPoint")
```

TypeError exception will be thrown. Hence, the following result will be displayed −

```
======================================================================
FAIL: testraiseRegex (__main__.raiseTest)
----------------------------------------------------------------------
Traceback (most recent call last):
   File "raisetest.py", line 11, in testraiseRegex
      self.assertRaisesRegexp(TypeError, "invalid", reg,123,"TutorialsPoint")
AssertionError: "invalid" does not match
   "first argument must be string or compiled pattern"
----------------------------------------------------------------------
```

UnitTest Framework - Time Test

Junit, the Java unit testing framework (Pyunit is implementation of JUnit) has a handy option of timeout. If a test takes more than specified time, it will be marked as failed.

Python's testing framework doesn't contain any support for time out. However, a third part module called timeout-decorator can do the job.

Download and install the module from −

https://pypi.python.org/packages/source/t/timeout-decorator/timeout-decorator-0.3.2.tar.gz

- Import timeout_decorator in the code
- Put timeout decorator before the test
- @timeout_decorator.timeout(10)

If a test method below this line takes more than the timeout mentioned (10 mins) here, a TimeOutError will be raised. For example −

```
import time
import timeout_decorator

class timeoutTest(unittest.TestCase):

   @timeout_decorator.timeout(5)
   def testtimeout(self):
```

```
      print "Start"
   for i in range(1,10):
      time.sleep(1)
      print "%d seconds have passed" % i

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

UnitTest Framework - Unittest2

unittest2 is a backport of additional features added to the Python testing framework in Python 2.7 and onwards. It is tested to run on Python 2.6, 2.7, and 3.*. Latest version can be downloaded from https://pypi.python.org/pypi/unittest2

To use unittest2 instead of unittest, simply replace import unittest with import unittest2.

Classes in unittest2 derive from the appropriate classes in unittest, so it should be possible to use the unittest2 test running infrastructure without having to switch all your tests to using unittest2 immediately. In case you intend to implement new features, subclass your testcase from **unittest2.TestCase** instead of unittest.TestCase

The following are the new features of unittest2 −

- **addCleanups** for better resource management
- Contains many new assert methods
- **assertRaises** as context manager, with access to the exception afterwards
- Has module level fixtures such as **setUpModule** and **tearDownModule**
- Includes **load_tests** protocol for loading tests from modules or packages
- **startTestRun** and **stopTestRun** methods on TestResult

In Python 2.7, you invoke the unittest command line features (including test discover) with **python -m unittest <args>**.

Instead, unittest2 comes with a script unit2.

```
unit2 discover
unit2 -v test_module
```

UnitTest Framework - Signal Handling

More efficient handling of control-C during a test run is provided by The -c/--catch command-line option to unittest, along with the **catchbreak** parameter. With catch break behavior enabled, control-C will allow the currently running test to complete, and the test run will then end and report all the results so far. A second control-c will raise a KeyboardInterrupt in the usual way.

If the unittest handler is called but signal.SIGINT handler isn't installed, then it calls for the default handler. This will normally be the expected behavior by code that replaces an installed handler and delegates to it. For individual tests that need unittest control-c handling disabled, the removeHandler() decorator can be used.

The following utility functions enable control-c handling functionality within test frameworks −

unittest.installHandler()

Install the control-c handler. When a **signal.SIGINT** is received all registered results have TestResult.stop() called.

unittest.registerResult(result)

Register a **TestResult** object for control-c handling. Registering a result stores a weak reference to it, so it doesn't prevent the result from being garbage collected.

unittest.removeResult(result)

Remove a registered result. Once a result has been removed then TestResult.stop() will no longer be called on that result object in response to a control-c.

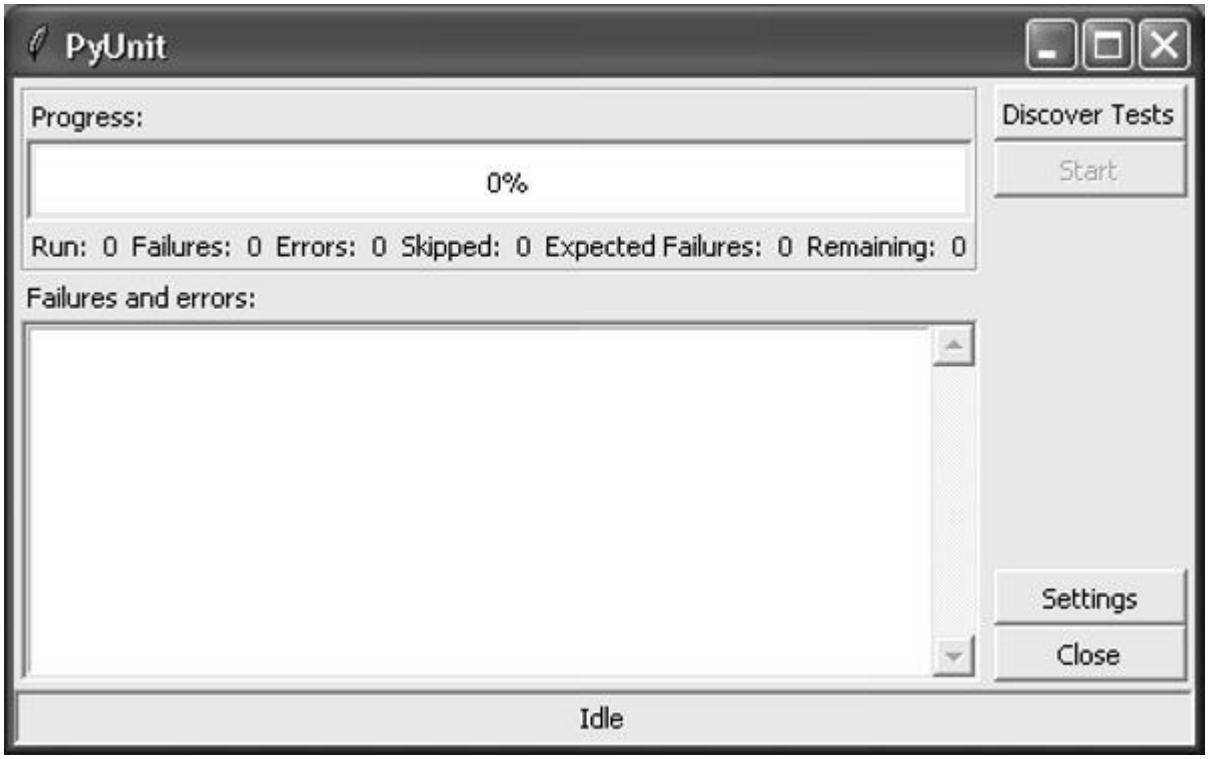
unittest.removeHandler(function = None)

When called without arguments, this function removes the control-c handler if it has been installed. This function can also be used as a test decorator to temporarily remove the handler whilst the test is being executed.

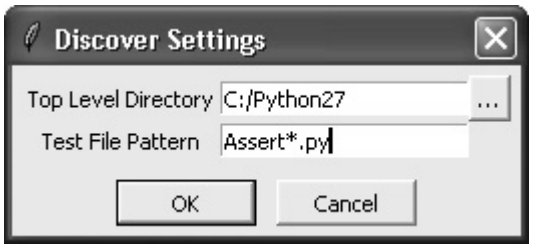
GUI Test Runner

The unittest module is installed to discover and run tests interactively. This utility, a Python script 'inittestgui.py' uses Tkinter module which is a Python port for TK graphics tool kit. It gives an easy to use GUI for discovery and running tests.

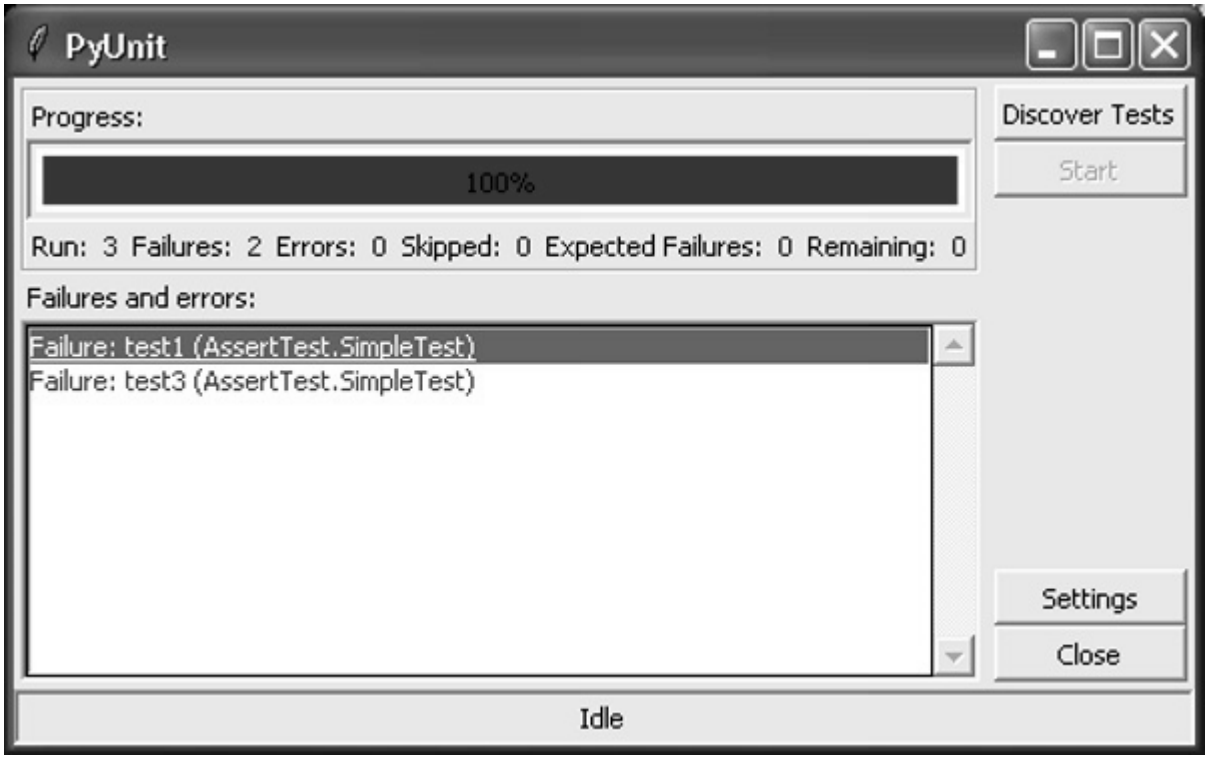
```
Python unittestgui.py
```



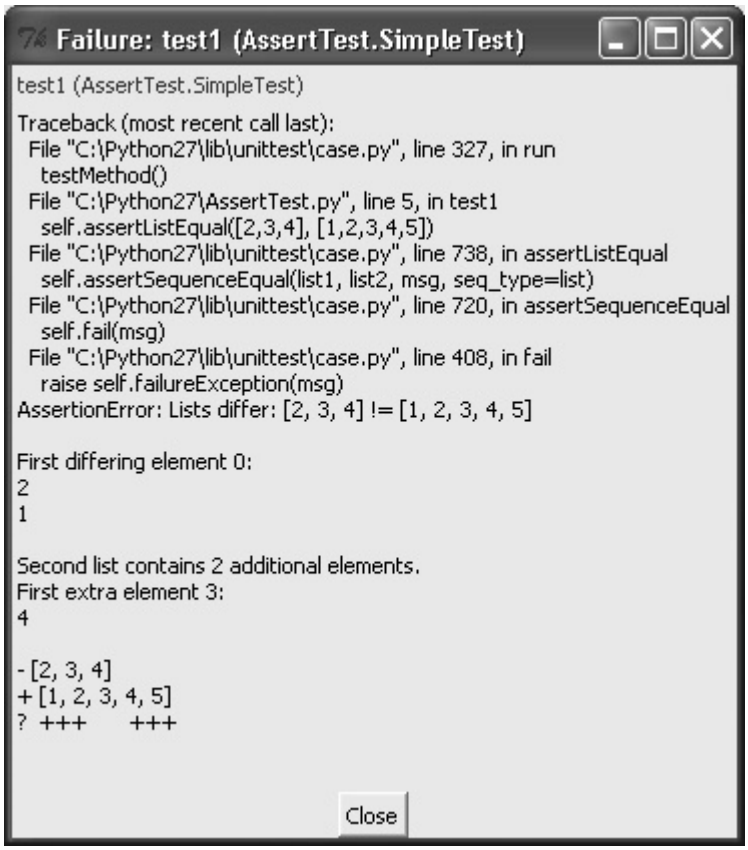
Click the 'Discover Tests' button. A small dialog box appears where you can select directory and modules from which test are to be run.



Finally, click the start button. Tests will be discovered from the selected path and module names, and the result pane will display the results.



In order to see the details of individual test, select and click on test in the result box −



If you do not find this utility in the Python installation, you can obtain it from the project page http://pyunit.sourceforge.net/ .

Similar, utility based on wxpython toolkit is also available there.

UnitTest Framework - Doctest

Python' standard distribution contains 'Doctest' module. This module's functionality makes it possible to search for pieces of text that look like interactive Python sessions, and executes these sessions to see if they work exactly as shown.

Doctest can be very useful in the following scenarios −

- To check that a module's docstrings are up-to-date by verifying that all interactive examples still work as documented.
- To perform regression testing by verifying that interactive examples from a test file or a test object work as expected.
- To write tutorial documentation for a package, liberally illustrated with input-output examples

In Python, a 'docstring' is a string literal which appears as the first expression in a class, function or module. It is ignored when the suite is executed, but it is recognized by the compiler and put into the **__doc__** attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

It is a usual practice to put example usage of different parts of Python code inside the docstring. The doctest module allows to verify that these docstrings are up-to-date with the intermittent revisions in code.

In the following code, a factorial function is defined interspersed with example usage. In order to verify if the example usage is correct, call the testmod() function in doctest module.

```
"""
This is the "example" module.

The example module supplies one function, factorial(). For example,

>>> factorial(5)
120
"""

def factorial(x):
   """Return the factorial of n, an exact integer >= 0.
   >>> factorial(-1)
   Traceback (most recent call last):
      ...
   ValueError: x must be >= 0
```

```
    """

    if not x >= 0:
       raise ValueError("x must be >= 0")
    f = 1
    for i in range(1,x+1):
       f = f*i
    return f

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Enter and save the above script as FactDocTest.py and try to execute this script from the command line.

```
Python FactDocTest.py
```

No output will be shown unless the example fails. Now, change the command line to the following −

```
Python FactDocTest.py –v
```

The console will now show the following output −

```
C:\Python27>python FactDocTest.py -v
Trying:
   factorial(5)
Expecting:
   120
ok
Trying:
   factorial(-1)
Expecting:
   Traceback (most recent call last):
      ...
   ValueError: x must be >= 0
ok
2 items passed all tests:
   1 tests in __main__
   1 tests in __main__.factorial
2 tests in 2 items.
2 passed and 0 failed.
```

```
Test passed.
```

If, on the other hand, the code of factorial() function doesn't give expected result in docstring, failure result will be displayed. For instance, change f = 2 in place of f = 1 in the above script and run the doctest again. The result will be as follows −

```
Trying:
   factorial(5)
Expecting:
   120
**********************************************************************
File "docfacttest.py", line 6, in __main__
Failed example:
factorial(5)
Expected:
   120
Got:
   240
Trying:
   factorial(-1)
Expecting:
   Traceback (most recent call last):
      ...
   ValueError: x must be >= 0
ok
1 items passed all tests:
   1 tests in __main__.factorial
**********************************************************************
1 items had failures:
   1 of 1 in __main__
2 tests in 2 items.
1 passed and 1 failed.
***Test Failed*** 1 failures.
```

Doctest: Checking Examples in a Text File

Another simple application of doctest is testing interactive examples in a text file. This can be done with the testfile() function.

The following text is stored in a text file named 'example.txt'.

```
Using ''factorial''
-------------------
This is an example text file in reStructuredText format. First import
```

```
''factorial'' from the ''example'' module:
   >>> from example import factorial
Now use it:
   >>> factorial(5)
   120
```

The file content is treated as docstring. In order to verify the examples in the text file, use the testfile() function of doctest module.

```
def factorial(x):
   if not x >= 0:
      raise ValueError("x must be >= 0")
   f = 1
   for i in range(1,x+1):
      f = f*i
   return f

if __name__ == "__main__":
   import doctest
   doctest.testfile("example.txt")
```

- As with the testmod(), testfile() won't display anything unless an example fails. If an example does fail, then the failing example(s) and the cause(s) of the failure(s) are printed to console, using the same format as testmod().
- In most cases a copy-and-paste of an interactive console session works fine, but doctest isn't trying to do an exact emulation of any specific Python shell.
- Any expected output must immediately follow the final '>>> ' or '... ' line containing the code, and the expected output (if any) extends to the next '>>> ' or all-whitespace line.
- Expected output cannot contain an all-whitespace line, since such a line is taken to signal the end of expected output. If expected output does contain a blank line, put <BLANKLINE> in your doctest example each place a blank line is expected.

UnitTest Framework - Doctest API

The doctest API revolves around the following two container classes used to store interactive examples from docstrings −

- **Example** − A single Python statement, paired with its expected output.
- **DocTest** − A collection of Examples, typically extracted from a single docstring or a text file.

The following additional processing classes are defined to find, parse, and run, and check doctest examples −

- **DocTestFinder** − Finds all docstrings in a given module, and uses a DocTestParser to create a DocTest from every docstring that contains interactive examples.
- **DocTestParser** − Creates a doctest object from a string (such as an object's docstring).
- **DocTestRunner** − Executes the examples in a doctest, and uses an OutputChecker to verify their output.
- **OutputChecker** − Compares the actual output from a doctest example with the expected output, and decides whether they match.

DocTestFinder Class

It is a processing class used to extract the doctests that are relevant to a given object, from its docstring and the docstrings of its contained objects. Doctests can currently be extracted from the following object types — modules, functions, classes, methods, staticmethods, classmethods, and properties.

This class defines the find() method. It returns a list of the DocTests that are defined by the *object's* docstring, or by any of its contained objects' docstrings.

DocTestParser Class

It is a processing class used to extract interactive examples from a string, and use them to create a DocTest object. This class defines the following methods −

- **get_doctest()** − Extract all doctest examples from the given string, and collect them into a **DocTest** object.
- **get_examples(string[, name])** − Extract all doctest examples from the given string, and return them as a list of **Example** objects. Line numbers are 0-based. The optional argument name is a name identifying this string, and is only used for error messages.
- **parse(string[, name])** − Divide the given string into examples and intervening text, and return them as a list of alternating **Examples** and strings. Line numbers for the **Examples** are 0-based. The optional argument name is a name identifying this string, and is only used for error messages.

DocTestRunner Class

This is a processing class used to execute and verify the interactive examples in a DocTest. The following methods are defined in it −

report_start()

Report that the test runner is about to process the given example. This method is provided to allow subclasses of **DocTestRunner** to customize their output; it should not be called directly

report_success()

Report that the given example ran successfully. This method is provided to allow subclasses of DocTestRunner to customize their output; it should not be called directly.

report_failure()

Report that the given example failed. This method is provided to allow subclasses of **DocTestRunner** to customize their output; it should not be called directly.

report_unexpected_exception()

Report that the given example raised an unexpected exception. This method is provided to allow subclasses of DocTestRunner to customize their output; it should not be called directly.

run(test)

Run the examples in *test* (a DocTest object), and display the results using the writer function *out*.

summarize([verbose])

Print a summary of all the test cases that have been run by this DocTestRunner, and return a *named tuple* TestResults(failed, attempted). The optional *verbose* argument controls how detailed the summary is. If the verbosity is not specified, then the DocTestRunner‘s verbosity is used.

OutputChecker Class

This class is used to check whether the actual output from a doctest example matches the expected output.

The following methods are defined in this class −

check_output()

Return **True** if the actual output from an example (*got*) matches with the expected output (*want*). These strings are always considered to match if they are identical; but depending on what option

flags the test runner is using, several non-exact match types are also possible. See section *Option Flags* and *Directives* for more information about option flags.

output_difference()

Return a string describing the differences between the expected output for a given example (*example*) and the actual output (*got*).

DocTest Integration with Unittest

The doctest module provides two functions that can be used to create unittest test suites from modules and text files containing doctests. To integrate with unittest test discovery, include a load_tests() function in your test module −

```
import unittest
import doctest
import doctestexample

def load_tests(loader, tests, ignore):
   tests.addTests(doctest.DocTestSuite(doctestexample))
   return tests
```

A combined TestSuite of tests from unittest as well as doctest will be formed and it can now be executed by unittest module's main() method or run() method.

The following are the two main functions for creating **unittest.TestSuite** instances from text files and modules with the doctests −

doctest.DocFileSuite()

It is used to convert doctest tests from one or more text files to a **unittest.TestSuite**. The returned unittest.TestSuite is to be run by the unittest framework and runs the interactive examples in each file. If any of the examples in a file fails, then the synthesized unit test fails, and a **failureException** exception is raised showing the name of the file containing the test and a (sometimes approximate) line number.

doctest.DocTestSuite()

It is used to convert doctest tests for a module to a **unittest.TestSuite**.

The returned unittest.TestSuite is to be run by the unittest framework and runs each doctest in the module. If any of the doctests fail, then the synthesized unit test fails, and a **failureException** exception is raised showing the name of the file containing the test and a (sometimes approximate) line number

Under the covers, DocTestSuite() creates a **unittest.TestSuite** out of doctest.DocTestCase instances, and DocTestCase is a subclass of unittest.TestCase.

Similarly, DocFileSuite() creates a unittest.TestSuite out of doctest.DocFileCase instances, and DocFileCase is a subclass of DocTestCase.

So both ways of creating a unittest.TestSuite run instances of DocTestCase. When you run doctest functions yourself, you can control the doctest options in use directly, by passing option flags to doctest functions.

However, if you're writing a unittest framework, unittest ultimately controls when and how the tests get run. The framework author typically wants to control doctest reporting options (perhaps, e.g., specified by command line options), but there's no way to pass options through unittest to doctest test runners.

UnitTest Framework - Py.test Module

It was in 2004 that Holger Krekel renamed his **std** package, whose name was often confused with that of the Standard Library that ships with Python, to the (only slightly less confusing) name 'py.' Though the package contains several sub-packages, it is now known almost entirely for its py.test framework.

The py.test framework has set up a new standard for Python testing, and has become very popular with many developers today. The elegant and Pythonic idioms it introduced for test writing have made it possible for test suites to be written in a far more compact style.

py.test is a no-boilerplate alternative to Python's standard unittest module. Despite being a fully-featured and extensible test tool, it boasts of a simple syntax. Creating a test suite is as easy as writing a module with a couple of functions.

py.test runs on all POSIX operating systems and WINDOWS (XP/7/8) with Python versions 2.6 and above.

Installation

Use the following code to load the pytest module in the current Python distribution as well as a py.test.exe utility. Tests can be run using both.

```
pip install pytest
```

Usage

You can simply use the assert statement for asserting test expectations. pytest's assert introspection will intelligently report intermediate values of the assert expression freeing you from the need to learn the many names of **JUnit legacy methods**.

```
# content of test_sample.py
def func(x):
   return x + 1

def test_answer():
   assert func(3) == 5
```

Use the following command line to run the above test. Once the test is run, the following result is displayed on console −

```
C:\Python27>scripts\py.test -v test_sample.py
============================= test session starts ====================
platform win32 -- Python 2.7.9, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- C:\Pyth
on27\python.exe
cachedir: .cache
rootdir: C:\Python27, inifile:
collected 1 items
test_sample.py::test_answer FAILED
================================= FAILURES ====================
_____________________________ test_answer ____________________________
   def test_answer():
>  assert func(3) == 5
E     assert 4 == 5
E     + where 4 = func(3)
test_sample.py:7: AssertionError
========================== 1 failed in 0.05 seconds ==================
```

The test can also be run from the command line by including pytest module using –m switch.

```
python -m pytest test_sample.py
```

Grouping Multiple Tests in a Class

Once you start to have more than a few tests it often makes sense to group tests logically, in classes and modules. Let's write a class containing two tests −

```
class TestClass:
   def test_one(self):
      x = "this"
      assert 'h' in x
   def test_two(self):
      x = "hello"
      assert hasattr(x, 'check')
```

The following test result will be displayed −

```
C:\Python27>scripts\py.test -v test_class.py
============================= test session starts ====================
platform win32 -- Python 2.7.9, pytest-2.9.1, py-1.4.31, pluggy-0.3.1 -- C:\Pyt
on27\python.exe
cachedir: .cache
rootdir: C:\Python27, inifile:
collected 2 items
test_class.py::TestClass::test_one PASSED
test_class.py::TestClass::test_two FAILED
================================= FAILURES ====================
_____________________________ TestClass.test_two _____________________________
self = <test_class.TestClass instance at 0x01309DA0>

   def test_two(self):
      x = "hello"
>  assert hasattr(x, 'check')
E     assert hasattr('hello', 'check')

test_class.py:7: AssertionError
==================== 1 failed, 1 passed in 0.06 seconds ======================
```

Nose Testing - Framework

The nose project was released in 2005, the year after **py.test** received its modern guise. It was written by Jason Pellerin to support the same test idioms that had been pioneered by py.test, but in a package that is easier to install and maintain.

The **nose** module can be installed with the help of pip utility

```
pip install nose
```

This will install the nose module in the current Python distribution as well as a nosetest.exe, which means the test can be run using this utility as well as using –m switch.

```
C:\python>nosetests –v test_sample.py
Or
C:\python>python –m nose test_sample.py
```

nose collects tests from **unittest.TestCase** subclasses, of course. We can also write simple test functions, as well as test classes that are not subclasses of unittest.TestCase. nose also supplies a number of helpful functions for writing timed tests, testing for exceptions, and other common use cases.

nose collects tests automatically. There's no need to manually collect test cases into test suites. Running tests is responsive, since **nose** begins running tests as soon as the first test module is loaded.

As with the unittest module, **nose** supports fixtures at the package, module, class, and test case level, so expensive initialization can be done as infrequently as possible.

Basic Usage

Let us consider nosetest.py similar to the script used earlier −

```
# content of nosetest.py
def func(x):
   return x + 1

def test_answer():
   assert func(3) == 5
```

In order to run the above test, use the following command line syntax −

```
C:\python>nosetests –v nosetest.py
```

The output displayed on console will be as follows −

```
nosetest.test_answer ... FAIL
======================================================================
FAIL: nosetest.test_answer
----------------------------------------------------------------------
Traceback (most recent call last):
   File "C:\Python34\lib\site-packages\nose\case.py", line 198, in runTest
      self.test(*self.arg)
```

```
   File "C:\Python34\nosetest.py", line 6, in test_answer
      assert func(3) == 5
AssertionError
----------------------------------------------------------------------
Ran 1 test in 0.000s
FAILED (failures = 1)
```

nose can be integrated with DocTest by using **with-doctest** option in athe bove command line.

```
\nosetests --with-doctest -v nosetest.py
```

You may use **nose** in a test script −

```
import nose
nose.main()
```

If you do not wish the test script to exit with 0 on success and 1 on failure (like unittest.main), use nose.run() instead −

```
import nose
result = nose.run()
```

The result will be true if the test run is successful, or false if it fails or raises an uncaught exception.

nose supports fixtures (setup and teardown methods) at the package, module, class, and test level. As with py.test or unittest fixtures, setup always runs before any test (or collection of tests for test packages and modules); teardown runs if setup has completed successfully, regardless of the status of the test run.

Nose Testing - Tools

The nose.tools module provides a number of testing aids that you may find useful, including decorators for restricting test execution time and testing for exceptions, and all of the same assertX methods found in unittest.TestCase.

- **nose.tools.ok_(expr, msg = None)** − Shorthand for assert.
- **nose.tools.eq_(a, b, msg = None)** − Shorthand for 'assert a == b, "%r != %r" % (a, b)
- **nose.tools.make_decorator(func)** − Wraps a test decorator so as to properly replicate metadata of the decorated function, including nose's additional stuff (namely, setup a

teardown).

- **nose.tools.raises(*exceptions)** − Test must raise one of expected exceptions to pass.
- **nose.tools.timed(limit)** − Test must finish within specified time limit to pass
- **nose.tools.istest(func)** − Decorator to mark a function or method as a test
- **nose.tools.nottest(func)** − Decorator to mark a function or method as not a test

Parameterized Testing

Python's testing framework, unittest, doesn't have a simple way of running parametrized test cases. In other words, you can't easily pass arguments into a **unittest.TestCase** from outside.

However, pytest module ports test parametrization in several well-integrated ways −

- **pytest.fixture()** allows you to define parametrization at the level of fixture functions.
- **@pytest.mark.parametrize** allows to define parametrization at the function or class level. It provides multiple argument/fixture sets for a particular test function or class.
- **pytest_generate_tests** enables implementing your own custom dynamic parametrization scheme or extensions.

A third party module 'nose-parameterized' allows Parameterized testing with any Python test framework. It can be downloaded from this link − https://github.com/wolever/nose-parameterized