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.

Live Demo

```
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 −

Sr.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
```