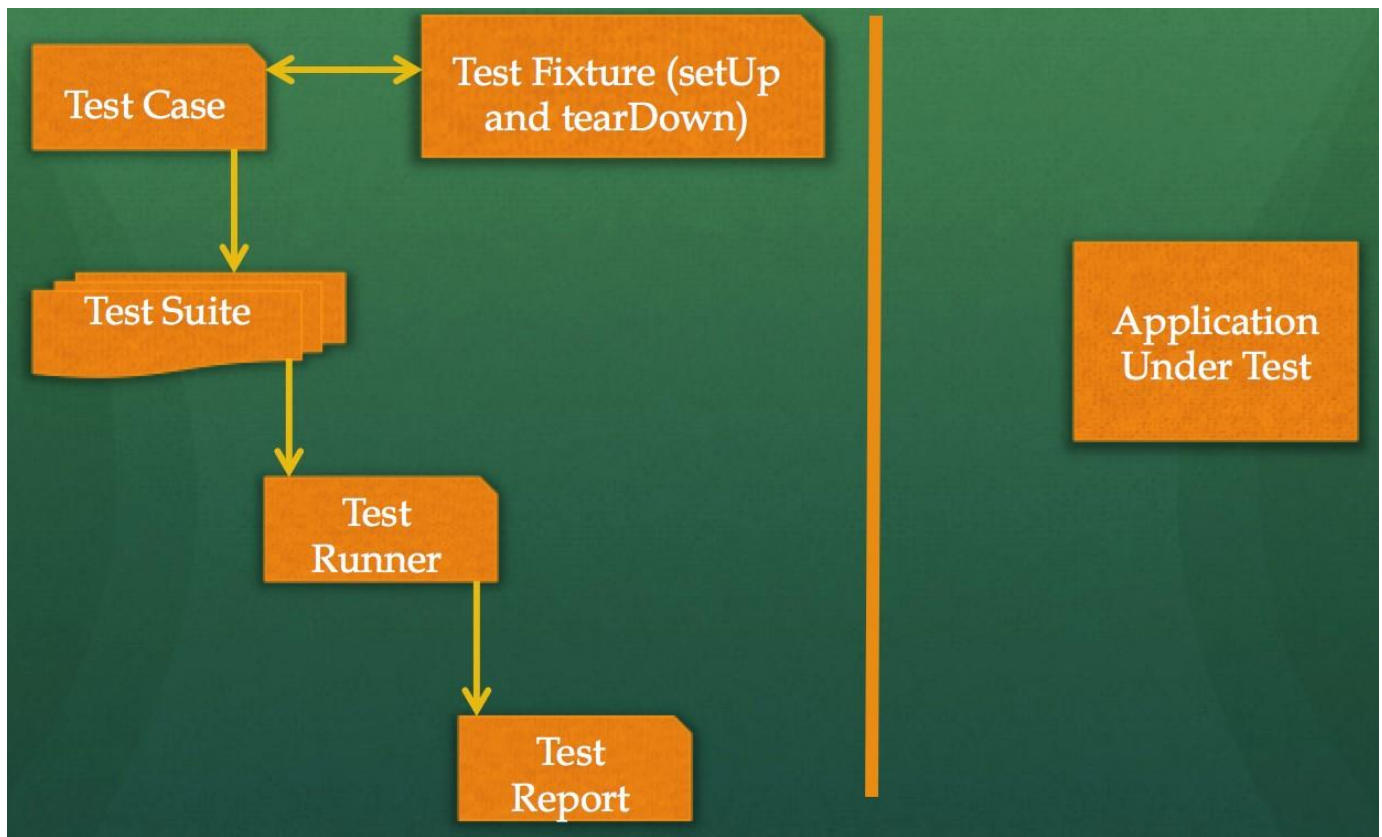


UnitTest Introduction



test fixture

A *test fixture* represents the preparation needed to perform one or more tests, and any associated cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

test case

A *test case* is the individual unit of testing. It 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

A *test suite* is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

test runner

A *test runner* 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.

Basic example

```
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is not a string
        with self.assertRaises(TypeError):
            s.split(2)

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

A testcase is created by subclassing [unittest.TestCase](#). The three individual tests are defined with methods whose names start with the letters test. This naming convention informs the test runner about which methods represent tests.

The order in which the various tests will be run is determined by sorting the test method names with respect to the built-in ordering for strings.

setUpClass() and **tearDownClass()** are run once for the whole class, **setUp()** and **tearDown()** will be executed before and after each test method

```
import unittest

def setUpModule():
    print("setup module")
def tearDownModule():
    print("teardown module")

class TestStringMethods(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        print("setUpClass")

    def setUp(self):
        print("setUpClass")
```

```

def test1(self):
    print("setUpClass")

def test2(self):
    print("test2")
def tearDown(self):
    print("tearDown")

    @classmethod
    def tearDownClass(cls):
        print("tearDownClass")

if __name__ == '__main__':
    unittest.main(verbosity=2)

```

Output:

```

setup module
setUpClass
setUp
tearDown
.setUp
tearDown
.tearDownClass
teardown module

```

Different module of setUp & tearDown hierarchy

```

# test_module.py
import unittest

class TestFoo(unittest.TestCase):
    @classmethod
    def setUpClass(self):
        print("foo setUpClass")
    @classmethod
    def tearDownClass(self):
        print("foo tearDownClass")
    def setUp(self):
        print("foo setUp")
    def tearDown(self):
        print("foo tearDown")
    def test_foo(self):
        self.assertTrue(True)

```

```

class TestBar(unittest.TestCase):
    def setUp(self):
        print("bar setUp")
    def tearDown(self):
        print("bar tearDown")
    def test_bar(self):
        self.assertTrue(True)

# test.py
from test_module import TestFoo
from test_module import TestBar
import test_module
import unittest

def setUpModule():
    print("setUpModule")

def tearDownModule():
    print("tearDownModule")

if __name__ == "__main__":
    test_module.setUpModule = setUpModule
    test_module.tearDownModule = tearDownModule
    suite1 = unittest.TestLoader().loadTestsFromTestCase(TestFoo)
    suite2 = unittest.TestLoader().loadTestsFromTestCase(TestBar)
    suite = unittest.TestSuite([suite1,suite2])
    unittest.TextTestRunner().run(suite)

```

Output

```

setUpModule
foo setUpClass
foo setUp
foo tearDown
.foo tearDownClass
bar setUp
bar tearDown
.tearDownModule

```

Run tests via unittest.TextTestRunner

```

>>> import unittest
>>> class TestFoo(unittest.TestCase):
...     def test_foo(self):

```

```

...         self.assertTrue(True)
...     def test_bar(self):
...         self.assertFalse(False)

>>> suite = unittest.TestLoader().loadTestsFromTestCase(TestFoo)
>>> unittest.TextTestRunner(verbosity=2).run(suite)
test_bar (__main__.TestFoo) ... ok
test_foo (__main__.TestFoo) ... ok

```

Skipping tests

1. `@unittest.skip(reason)`
2. `@unittest.skipIf(condition, reason)`
3. `@unittest.skipUnless(condition, reason)`
4. `@unittest.expectedFailure`
5. `exception unittest.SkipTest(reason)`

```

class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3), "not supported in this library version")
    def test_format(self):
        # Tests that work for only a certain version of the library.
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_windows_support(self):
        # windows specific testing code
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available")
        # test code that depends on the external resource
        pass

```

Add an expected failure test

```

import unittest

class MyTest(unittest.TestCase):

    @unittest.expectedFailure
    def runTest(self):
        self.assertEqual(0, 1)

unittest.TextTestRunner().run(MyTest())

```

Group multiple testcases into a suite

```
>>> import unittest
>>> class TestFooBar(unittest.TestCase):
...     def test_foo(self):
...         self.assertTrue(True)
...     def test_bar(self):
...         self.assertTrue(True)
...
>>> class TestHelloWorld(unittest.TestCase):
...     def test_hello(self):
...         self.assertEqual("Hello", "Hello")
...     def test_world(self):
...         self.assertEqual("World", "World")
...
>>> suite_loader = unittest.TestLoader()
>>> suite1 = suite_loader.loadTestsFromTestCase(TestFooBar)
>>> suite2 = suite_loader.loadTestsFromTestCase(TestHelloWorld)
>>> suite = unittest.TestSuite([suite1, suite2])
>>> unittest.TextTestRunner(verbosity=2).run(suite)
```

Group multiple tests from different TestCase

```
>>> suite = unittest.TestSuite()
>>> suite.addTest(TestFoo('test_foo'))
>>> suite.addTest(TestBar('test_bar'))
>>> unittest.TextTestRunner(verbosity=2).run(suite)
```

Loading tests to suite and running tests

```
allTests = unittest.TestLoader()
```

The TestLoader() class is used to create test suites from classes and modules in the python project

```
allTests = unittest.TestLoader()
suite = allTests.loadTestsFromTestCase(TestBing, TestCherCherTech)
```

loadTestsFromTestCase(TestClassName) method loads all the test cases from the given class, a test case instance is created for the method getTestCaseNames() which notifies each test cases present in the Test Class.

```
suite = TestSuite()
# load the tests
tests = unittest.TestLoader()

# add the tests to the suite
```

```

suite.addTests(tests.loadTestsFromTestCase(TestBing))
suite.addTests(tests.loadTestsFromTestCase(TestCherCherTech))

# run the suite
runner = unittest.TextTestRunner()
runner.run(suite)

```

TextTestRunner class is the basic runner class present python, after loading all the methods using TestLoader and loadTestsFromTestCase, we have to start the run using run method present in the TextTestRunner class.

```

import unittest
class Test_Me(unittest.TestCase):
    def test_Java(self):
        print("Java11")

    def test_Python(self):
        print("Python")

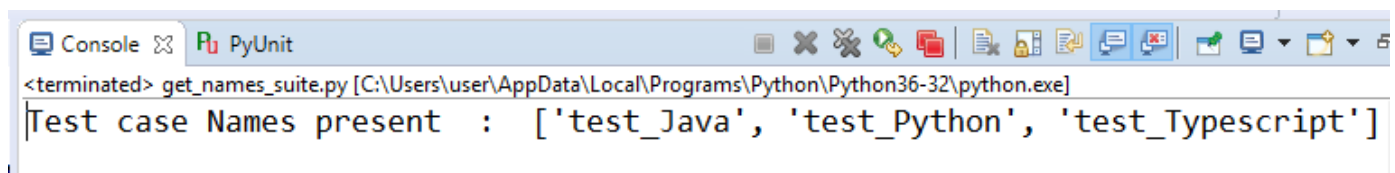
    def test_Typescript(self):
        print("Typescript")
if __name__ == "__main__":
    # load the tests
    tests = unittest.TestLoader()

    # add the tests to the suite
    alltestNames = tests.getTestCaseNames(Test_Me)

    # print test method present in test class
    print("Test case Names present : " , alltestNames)

```

getTestCaseNames(testCaseClass) method returns a sorted sequence of method names found within testCaseClass; test case class should be a subclass of TestCase



The screenshot shows a console window titled 'PyUnit' with the following output:

```

<terminated> get_names_suite.py [C:\Users\user\AppData\Local\Programs\Python\Python36-32\python.exe]
Test case Names present : ['test_Java', 'test_Python', 'test_Typescript']

```

loadTestsFromModule() :

```

*****File1 module1*****
import unittest
class Test_Module1(unittest.TestCase):
    def test_module1_WELCOME(self):
        print("module1 - welcome")

*****File2 Module2*****
import unittest
class TestModule2_HI(unittest.TestCase):
    def test_module2_hi(self):
        print("module2 - hi")
class TestModule2_HELLO(unittest.TestCase):

    def test_module2_hello(self):
        print("module2 - hello")

```

```

*****File3 runner-module*****
from unittest.suite import TestSuite
import unittest
from pythontest import module1, module2
if __name__ == "__main__":
    # create the suite from the test classes
    suite = TestSuite()
    # load the tests
    tests = unittest.TestLoader()
    # add the tests to the suite from modules
    suite.addTests(tests.loadTestsFromModule(module1))
    suite.addTests(tests.loadTestsFromModule(module2))

    # run the suite
    runner = unittest.TextTestRunner()
    runner.run(suite)
*****File1 module1*****
import unittest
class Test_Module1(unittest.TestCase):
    def test_module1_WELCOME(self):
        print("module1 - welcome")

*****File2 Module2*****
import unittest
class TestModule2_HI(unittest.TestCase):
    def test_module2_hi(self):
        print("module2 - hi")
class TestModule2_HELLO(unittest.TestCase):

    def test_module2_hello(self):
        print("module2 - hello")

*****File3 runner-module*****
from unittest.suite import TestSuite
import unittest
from pythontest import module1, module2
if __name__ == "__main__":
    # create the suite from the test classes
    suite = TestSuite()
    # load the tests
    tests = unittest.TestLoader()
    # add the tests to the suite from modules
    suite.addTests(tests.loadTestsFromModule(module1))
    suite.addTests(tests.loadTestsFromModule(module2))

    # run the suite
    runner = unittest.TextTestRunner()
    runner.run(suite)

```



```
Console x PyUnit
<terminated> runner_module.py [C:\Users\user\AppData\Local\Programs\Python\Python36-32\python.exe]
module1 - welcome
.module2 - hello
module2 - hi
..
-----
Ran 3 tests in 0.001s

OK
```

Create Test Suite

addTest(test) :

```
import unittest
from unittest.suite import TestSuite
class Test_Tou(unittest.TestCase):
    def test_touch(self):
        print("touchme")

class Test_Ting(unittest.TestCase):
    def test_tingle(self):
        print("tingle me")
if __name__ == "__main__":
    print("Running using test suite with add test")
    # create the suite from the test classes
    suite = TestSuite()
    # load the test from test class
    suite.addTest(Test_Tou("test_touch"))

    # run the suite
    runner = unittest.TextTestRunner()
    runner.run(suite)
```

addTests(tests) :

```
import unittest
from unittest.suite import TestSuite
from unittest.loader import TestLoader
from unittest.runner import TextTestRunner
class Test_Tou(unittest.TestCase):
    def test_touch(self):
        print("touchme")
    def test_tingle(self):
        print("tingle me")
if __name__ == "__main__":
    print("Running using test suite with add tests")
    # create the suite from the test classes
    suite = TestSuite()
    # will contains all tests
    tests = TestLoader()
    # load the test from test class
    suite.addTests(tests.loadTestsFromTestCase(Test_Tou))
```

```
# run the suite
runner = TextTestRunner()
runner.run(suite)
```

countTestCases() :

```
# create the suite from the test classes
suite = TestSuite()
# will contains all tests
tests = TestLoader()
# load the test from test class
suite.addTests(tests.loadTestsFromTestCase(Test_Tou))
# number of test cases present
print("No. of test cases present : ", suite.countTestCases())
```

Method	Checks that
<code>assertEqual(a, b)</code>	<code>a == b</code>
<code>assertNotEqual(a, b)</code>	<code>a != b</code>
<code>assertTrue(x)</code>	<code>bool(x) is True</code>
<code>assertFalse(x)</code>	<code>bool(x) is False</code>
<code>assertIs(a, b)</code>	<code>a is b</code>
<code>assertIsNot(a, b)</code>	<code>a is not b</code>
<code>assertIsNone(x)</code>	<code>x is None</code>
<code>assertIsNotNone(x)</code>	<code>x is not None</code>
<code>assertIn(a, b)</code>	<code>a in b</code>
<code>assertNotIn(a, b)</code>	<code>a not in b</code>
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>
<code>assertAlmostEqual(a, b)</code>	<code>round(a-b, 7) == 0</code>
<code>assertNotAlmostEqual(a, b)</code>	<code>round(a-b, 7) != 0</code>
<code>assertGreater(a, b)</code>	<code>a > b</code>
<code>assertGreaterEqual(a, b)</code>	<code>a >= b</code>
<code>assertLess(a, b)</code>	<code>a < b</code>
<code>assertLessEqual(a, b)</code>	<code>a <= b</code>

Method	Checks that
<code>assertRegex(s, r)</code>	<code>r.search(s)</code>
<code>assertNotRegex(s, r)</code>	<code>not r.search(s)</code>
<code>assertCountEqual(a, b)</code>	a and b have the same elements in the same number, regardless of their order.