The Python Unittest Framework

Musa Baloyi

July 13, 2018

Table of contents

The origins of automated testing

- Kent Beck designed and wrote SUnit for Smalltalk.
- Automated testing became a thing when Kent Beck and Erich Gamma introduced JUnit for Java.
- Eventually, it was ported to almost every programming language.
- Now most of the programming languages come pre-packaged with at least one xUnit-style test automation library.
- xUnit is the collective name for several unit testing frameworks for various languages.
- All the xUnit-style unit testing frameworks more or less derive their functionality, structure, and coding style from SUnit.

The history of automated testing in Python

- A Python version, originally dubbed PyUnit, was created in 1999 and added to Python's standard set of libraries later in 2001.
- PyUnit became part of the Python Standard library from version 2.5 onward.
- PyUnit was the Python port from JUnit.
- The PyUnit library is available for both major versions of Python, 2.7 and 3.x.
- unittest came to life as a third-party module PyUnit.
- Since then, the Python community has referred to it as unittest, the name of the library imported into the test code.
- Unittest is the batteries-included test automation library of Python, which requires no extra installation steps.

Other testing frameworks in Python

- Many programming languages like Python and Java have more than one xUnit-style framework.
- Java has TestNG in addition to JUnit. Python has nose, pytest, and Nose2 apart from unittest.
- docstring and doctest and their use in writing simple, static, yet elegant test cases for Python 3 programs.
- Due to the lack of features like API, configurable tests, and test fixtures, doctest enjoys very limited popularity.

Major components of the architecture of xUnit-style test automation libraries:

1. Test case class
2. Test fixtures
3. Assertions
4. Test suite
5. Test runner
6. Test result formatter

Entry point of unittest is the TestCase class

```
1  import unittest
2
3  class TestClass01(unittest.TestCase):
4
5      def test_case01(self):
6          my_str = "ASHWIN"
7          my_int = 99
8
9          self.assertTrue(isinstance(my_str, str))
10         self.assertTrue(isinstance(my_int, int))
11
12     def test_case02(self):
13         my_pi = 3.14
14         self.assertFalse(isinstance(my_pi, int))
15
16
17 if __name__ == '__main__':
18     unittest.main()
```

- python3 test_module01.py

Order of execution of the test methods

```
1  import unittest
2  import inspect
3
4  class TestClass02(unittest.TestCase):
5
6      def test_case02(self):
7          print("\nRunning Test Method: " + inspect.stack()[0][3])
8
9      def test_case01(self):
10         print("\nRunning Test Method: " + inspect.stack()[0][3])
11
12 if __name__ == '__main__':
13     unittest.main()
```

- python3 test_module02.py

Verbosity control

```
1  import unittest
2  import inspect
3
4  def add(x, y):
5      print("We're in custom made function: " + inspect.stack()[0][3])
6      return(x+y)
7
8  class TestClass03(unittest.TestCase):
9
10     def test_case01(self):
11         print("\nRunning Test Method: " + inspect.stack()[0][3])
12         self.assertEqual(add(2,3), 5)
13
14     def test_case02(self):
15         print("\nRunning Test Method: " + inspect.stack()[0][3])
16         my_var = 3.14
17         self.assertTrue(isinstance(my_var, float))
18
19     def test_case03(self):
20         print("\nRunning Test Method: " + inspect.stack()[0][3])
21         self.assertEqual(add(2,2), 5)
22
23     def test_case04(self):
24         print("\nRunning Test Method: " + inspect.stack()[0][3])
25         my_var = 3.14
26         self.assertTrue(isinstance(my_var, int))
27
28 if __name__ == '__main__':
29     unittest.main(verbosity=2)
```

- python3 test_module01.py -v
- python3 test_module03.py

Multiple test classes within the same test file

```
1  import unittest
2  import inspect
3
4  def add(x, y):
5      print("We're in custom made function: " + inspect.stack()[0][3])
6      return(x+y)
7
8  class TestClass04(unittest.TestCase):
9
10     def test_case01(self):
11         print("\nClassname: " + self.__class__.__name__)
12         print("\nRunning Test Method: " + inspect.stack()[0][3])
13
14 class TestClass05(unittest.TestCase):
15
16     def test_case01(self):
17         print("\nClassname: " + self.__class__.__name__)
18         print("\nRunning Test Method: " + inspect.stack()[0][3])
19
20 if __name__ == '__main__':
21     unittest.main(verbosity=2)
```

- python3 test_module04.py

Test fixtures

```python
import unittest

def setUpModule():
    """ called once, before anything else in this module """
    print("In setUpModule()...")

def tearDownModule():
    """ called once, after anything else in this module """
    print("In tearDownModule()...")


class TestClass06(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        """ called once, before any test """
        print("In setUpClass()...")

    @classmethod
    def tearDownClass(cls):
        """ called once, after all tests, if setUpClass successful """
        print("In tearDownClass()...")

    def setUp(self):
        """ called multiple times, before every test method """
        print("\nIn setUp()...")

    def tearDown(self):
        """ called multiple times, after every test method """
        print("\nIn tearDown()...")

    def test_case01(self):
        self.assertTrue("PYTHON".isupper())
        print("In test_case01()")

    def test_case02(self):
        self.assertFalse("python".isupper())
        print("In test_case02()")

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

- python3 test_module05.py

Running without unittest.main()

```
1 import unittest
2
3 class TestClass07(unittest.TestCase):
4
5     def test_case01(self):
6         self.assertTrue("PYTHON".isupper())
7         print("\nIn test_case01()")
```

- python3 -m unittest test_module06

Controlling the granularity of test execution

```
1  import unittest
2  import inspect
3
4  def add(x, y):
5      print("We're in custom made function: " + inspect.stack()[0][3])
6      return(x+y)
7
8  class TestClass04(unittest.TestCase):
9
10     def test_case01(self):
11         print("\nClassname: " + self.__class__.__name__)
12         print("\nRunning Test Method: " + inspect.stack()[0][3])
13
14 class TestClass05(unittest.TestCase):
15
16     def test_case01(self):
17         print("\nClassname: " + self.__class__.__name__)
18         print("\nRunning Test Method: " + inspect.stack()[0][3])
19
20 if __name__ == '__main__':
21     unittest.main(verbosity=2)
```

- python3 -m unittest -v test_module04.TestClass04

Command line options

```
1  import unittest
2
3
4  class TestClass08(unittest.TestCase):
5
6      def test_case01(self):
7          self.assertTrue("PYTHON".isupper())
8          print("\nIn test_case01()")
9
10     def test_case02(self):
11         self.assertTrue("Python".isupper())
12         print("\nIn test_case02()")
13
14     def test_case03(self):
15         self.assertTrue(True)
16         print("\nIn test_case03()")
17
18 if __name__ == '__main__':
19     unittest.main(verbosity=2)
```

- python3 -m unittest -q test_module07
- python3 -m unittest -fv test_module07
- python3 -m unittest -h

Placing the dev and test code in A SINGLE dir

```
1  import unittest
2  import test.test_me as test_me
3  #from test import test_me
4
5  class TestClass09(unittest.TestCase):
6
7      def test_case01(self):
8          self.assertEqual(test_me.add(2, 3), 5)
9          print("\nIn test_case01()")
10
11     def test_case02(self):
12         self.assertEqual(test_me.mul(2, 3), 6)
13         print("\nIn test_case02()")
```

- python3 -m unittest -v test_module08

Placing the dev and test code in SEPARATE dirs

```python
from mypackage.mymathlib import *
import unittest

math_obj = 0


def setUpModule():
    """called once, before anything else in the module"""
    print("In setUpModule()...")
    global math_obj
    math_obj = mymathlib()


def tearDownModule():
    """called once, after everything else in the module"""
    print("In tearDownModule()...")
    global math_obj
    del math_obj


class TestClass10(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        """called only once, before any test in the class"""
        print("In setUpClass()...")

    def setUp(self):
        """called once before every test method"""
        print("\nIn setUp()...")

    def test_case01(self):
        print("In test_case01()")
        self.assertEqual(math_obj.add(2, 5), 7)

    def test_case02(self):
        print("In test_case02()")

    def tearDown(self):
        """called once after every test method"""
        print("In tearDown()...")

    @classmethod
    def tearDownClass(cls):
        """called once, after all the tests in the class"""
        print("In tearDownClass()...")

```

- python3 -m unittest test_module09

Other useful methods

```
1 import unittest
2
3 class TestClass11(unittest.TestCase):
4
5     def test_case01(self):
6         """ This is a test method """
7         print("\nIn test_case01()")
8         print(self.id())
9         print(self.shortDescription())
```

- python3 -m unittest -v test_module10

Failing a test

```
1 import unittest
2
3 class TestClass12(unittest.TestCase):
4
5     def test_case01(self):
6         """ This is a test method """
7         print(self.id())
8         self.fail()
```

- python3 -m unittest -v test_module11

Skipping tests

Decorators are used for skipping tests conditionally or unconditionally:

- unittest.skip(reason)
- unittest.skipIf(condition, reason)
- unittest.skipUnless(condition, reason)
- unittest.expectedFailure()

Skipping tests: examples

```
1  import sys
2  import unittest
3
4  class TestClass13(unittest.TestCase):
5
6      @unittest.skip("Demonstrating unconditional skipping")
7      def test_case01(self):
8          self.fail("FATAL")
9
10     @unittest.skipUnless(sys.platform.startswith("win"), "Requires
       Windows")
11     def test_case02(self):
12         # Windows specific code
13         pass
14
15     @unittest.skipUnless(sys.platform.startswith("linux"), "Requires
       Linux")
16     def test_case03(self):
17         # Linux specific code
18         pass
19
20     @unittest.skipUnless(sys.platform.startswith("dar"), "Requires
       Darwin")
21     def test_case04(self):
22         # Linux specific code
23         pass
```

- python3 -m unittest -v test_module12

Exceptions in the test case

```
1 import unittest
2
3 class TestClass14(unittest.TestCase):
4
5     def test_case01(self):
6         raise Exception
```

- python3 -m unittest -v test_module13

assertRaises()

```python
import unittest

class Calculator:

    def add1(self, x, y):
        return x+y

    def add2(self, x, y):
        number_types = (int, float, complex)
        if isinstance(x, number_types) and isinstance(y, number_types):
            return x+y
        else:
            raise ValueError

calc = 0

class TestClass16(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        global calc
        calc = Calculator()

    def setUp(self):
        print("\nIn setUp()...")

    def test_case01(self):
        self.assertEqual(calc.add1(2,2), 4)

    def test_case02(self):
        self.assertEqual(calc.add2(2,2),4)

    def test_case03(self):
        self.assertRaises(ValueError, calc.add1, 2, "two")

    def test_case04(self):
        self.assertRaises(ValueError, calc.add2, 2, "two")

    def tearDown(self):
        print("\nIn tearDown()...")

    @classmethod
    def tearDownClass(cls):
        global calc
        del calc

```

- python3 test_module14.py

References

1. Python Unit Test Automation: Practical Techniques for Python Developers and Testers by Ashwin Pajankar (2017)
2. Python Testing Cookbook by Bhaskar N. Das , Greg L. Turnquist (2018)
3. Unit testing framework (https://docs.python.org/3/library/unittest.html)
4. Python Testing with pytest by Brian Okken (2017)

Assertions in unittest

Method	Checks That
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

More assertions in unittest

Method	Checks That
assertAlmostEqual(a, b)	round(a-b, 7) == 0
assertNotAlmostEqual(a, b)	round(a-b, 7) != 0
assertGreater(a, b)	a > b
assertGreaterEqual(a, b)	a >= b
assertLess(a, b)	a < b
assertLessEqual(a, b)	a <= b
assertRegexpMatches(s, r)	r.search(s)
assertNotRegexpMatches(s, r)	not r.search(s)
assertItemsEqual(a, b)	sorted(a) == sorted(b)
assertDictContainsSubset(a, b)	all the key/value pairs in a exist in b

Even more assertions in unittest

Method	Used to Compare
assertMultiLineEqual(a, b)	strings
assertSequenceEqual(a, b)	sequences
assertListEqual(a, b)	lists
assertTupleEqual(a, b)	tuples
assertSetEqual(a, b)	sets or frozensets
assertDictEqual(a, b)	dicts

py.test vs unittest

Property	py.test	unittest
Installation	Easy installation with pip	Part of standard library (None required)
Color coding	Green for PASS, Red for FAILURE	No color coding (External library required)
Paradigm	Functional and object-oriented	Object-oriented (OOP) only
Interoperability	Can run unittest and nose tests	Lower level
Test runner	py.test; -m pytest	python; -m unittest; py.test; nose
Usage	Command line flag	import statement
Test discovery	Through py.test	Through python -m unittest discover
Verbosity	Through -v	Through -v
Fixtures	xUnit and custom	xUnit (Decorators can be function calls)
Granularity	Down to test method	Down to test method
Assertions	Through assert	Through assert<Equality, Truth, DataType, etc>
Exceptions	pytest.raises()	assertRaises()
Command line	Much more extensive options	Extensive options

Table: Let's battle it out!!!