TDD in Python in 5 minutes

by Giorgio Sironi MVB · Feb. 28, 12 · Web Dev Zone

Test-Driven Development is a basic technique nowadays, that you adapt to a new language in the same way as you learn the syntax of iterations (or recursions) or of function calls. Here is my take on transporting my Java and PHP TDD experience into Python.

The basics

The Python official interpreter ships a unittest module, that you can use in substitution of xUnit tools from other languages. Tests built for unittest are classes **extending unittest.TestCase**.

By convention, **methods starting with *test_*** are recognized as test to be run, while setUp() and tearDown() are reserved names for routines to execute once for each test, respectively at the start and at the end of it as you would expect.

Each of **these methods take only *self*** as a parameter, which means they will be actually called with no arguments. You can share references between setUp, tearDown and test_* methods via self, which is the Python equivalent of this.

However, you're not obliged to define fields in the class's body, as you can assign new ones to self at any time. This example from the manual also contains a __main__ function to run a test file by itself, which is not really necessary if you use **python -m unittest**.

```
import random
import unittest

class TestSequenceFunctions(unittest.TestCase):

    def setUp(self):
        self.seq = range(10)

    def test_shuffle(self):
        # make sure the shuffled sequence does not lose any elements
        random.shuffle(self.seq)
        self.seq.sort()
        self.assertEqual(self.seq, range(10))

        # should raise an exception for an immutable sequence
```

```
        self.assertRaises(TypeError, random.shuffle, (1,2,3))

    def test_choice(self):
        element = random.choice(self.seq)
        self.assertTrue(element in self.seq)

    def test_sample(self):
        with self.assertRaises(ValueError):
            random.sample(self.seq, 20)
        for element in random.sample(self.seq, 5):
            self.assertTrue(element in self.seq)

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

Assertions

Apart from the basic methods structure, unittest also features assertion methods inherited from TestCase as the main way to check the behavior of code.

- **assertEqual(expected, actual)** is the equivalent of assertEquals() and lets you specify an expected value along with an actual one obtained. Python's equality for objects is based on the __eq__ method.
- **assertNotEqual(notExpected, actual)** is the opposite of the previous assertion.
- **assertTrue(expression) and assertFalse(expression)** allows you to create custom assertions; expression is a boolean value obtained with <, >, other comparison operators or methods, or the combination of other booleans with *and*, *or*, and *not*.
- **assertIsInstance(object, class)** checks object is the instance of class or of a subclass.

The generation of Test Doubles such as Stubs and Mocks is not supported by default, but there are many libraries you can integrate for behavior-based testing.

Running

The files containing test cases should start with the test* prefix (like test_tennis.py), so that they can be found automatically:

```
python -m unittest discover
```

In unittest conventions, it is not necessary to map a test case class to a single: maybe it is more natural to map the tests for a module to a single file, as modules can contain many decoupled functions instead of classes. What you test in a file/test case class/method depends only on what you import and instantiate, not on restriction from the framework.

Thus file filtering can be applied to run only a file (tests for a module), only a class, or only a test

Thus the filtering can be applied to run only a file (tests for a module), only a class, or only a test method:

```
python -m unittest test_random
python -m unittest test_random.TestSequenceFunctions
python -m unittest test_random.TestSequenceFunctions.test_shuffle
```

A kata

To try all these tools on the field, I executed the tennis kata. It consists of implementing the scoring rules of a tennis set:

1. Each player can have either of these points in one game, described as 0-15-30-40. Each time a player scores, it advances of one position in the scale.
2. A player at 40 who scores wins the set. Unless...
3. If both players are at 40, we are in a *deuce*. If the game is in deuce, the next scoring player will gain an *advantage*. Then if the player in advantage scores he wins, while if the player not in advantage scores they are back at deuce.

The final result (of the test) is:

```
from tennis import Set, Scores
from unittest import TestCase

class TestSetWinning(TestCase):
    def test_score_grows(self):
        set = Set()
        self.assertEqual("0", set.firstScore())
        set.firstScores()
        self.assertEqual("15", set.firstScore())
        self.assertEqual("0", set.secondScore())
        set.secondScores()
        self.assertEqual("15", set.secondScore())
    def test_player_1_win_when_scores_at_40(self):
        set = Set()
        set.firstScores(3)
        self.assertEqual(None, set.winner())
        set.firstScores()
        self.assertEqual(1, set.winner())
    def test_player_2_win_when_scores_at_40(self):
        set = Set()
        set.secondScores(3)
        self.assertEqual(None, set.winner())
        set.secondScores()
```

```
        self.assertEqual(2, set.winner())
    def test_deuce_requires_1_more_than_one_ball_to_win(self):
        set = Set()
        set.firstScores(3)
        set.secondScores(3)
        set.firstScores()
        self.assertEqual(None, set.winner())
        set.firstScores()
        self.assertEqual(1, set.winner())
    def test_deuce_requires_2_more_than_one_ball_to_win(self):
        set = Set()
        set.firstScores(3)
        set.secondScores(3)
        set.secondScores()
        self.assertEqual(None, set.winner())
        set.secondScores()
        self.assertEqual(2, set.winner())
    def test_player_can_return_to_deuce_by_scoring_against_the_others_advantage(sel
        set = Set()
        set.firstScores(3)
        set.secondScores(3)
        self.assertEqual(None, set.winner())
        set.firstScores()
        set.secondScores()
        set.firstScores()
        set.secondScores()
        self.assertEqual(None, set.winner())
        self.assertEqual("40", set.firstScore())
        self.assertEqual("40", set.secondScore())

class TestScoreNames(TestCase):
    def test_score_names(self):
        scores = Scores()
        self.assertEqual("0", scores.scoreName(0))
        self.assertEqual("15", scores.scoreName(1))
        self.assertEqual("30", scores.scoreName(2))
        self.assertEqual("40", scores.scoreName(3))
        self.assertEqual("A", scores.scoreName(4))
```