Python | Main course

Session 24

Introduction

Python unittest

by Mohammad Amin H.B. Tehrani

www.maktabsharif.ir

Introduction

Software testing

Software Testing is a method to check whether the actual software product matches expected requirements and to ensure that software product is Defect free.

It involves execution of software/system components using manual or automated tools to evaluate one or more properties of interest. The purpose of software testing is to identify errors, gaps or missing requirements in contrast to actual requirements.

Why Software Testing is Important?

Software Testing is Important because if there are any bugs or errors in the software, it can be identified early and can be solved before delivery of the software product. Properly tested software product ensures reliability, security and high performance which further results in time saving, cost effectiveness and customer satisfaction.

Examples

Software bugs can potentially cause monetary and human loss, and history is full of such examples.

- Nissan cars recalled over 1 million cars from the market due to software failure in the airbag sensory detectors. There has been reported two accident due to this software failure.
- Starbucks was forced to close about 60 percent of stores in the U.S and Canada due to software failure in its POS system. At one point, the store served coffee for free as they were unable to process the transaction.
- Some of Amazon's third-party retailers saw their product price is reduced to 1p due to a software glitch. They were left with heavy losses.
- In 2015 fighter plane F-35 fell victim to a software bug, making it unable to detect targets correctly.
- China Airlines Airbus A300 crashed due to a software bug on April 26, 1994, killing 264 innocents live
- In April of 1999, a software bug caused the failure of a $1.2 billion military satellite launch, the costliest accident in history
- In May of 1996, a software bug caused the bank accounts of 823 customers of a major U.S. bank to be credited with 920 million US dollars.

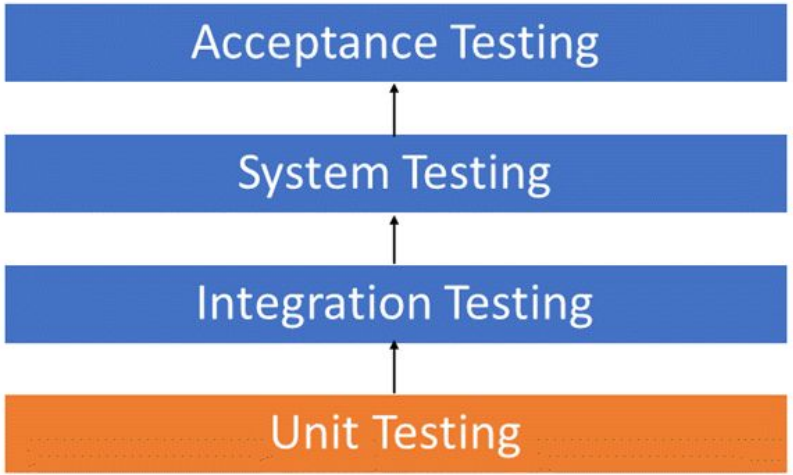
Testing categories

Testing Category	Types of Testing
Functional Testing	Unit Testing Integration Testing UAT (User Acceptance Testing) Localization Globalization
Non-Functional Testing	Performance Load Volume Scalability Usability
Maintenance	Regression Maintenance

Unit testing

UNIT TESTING is a **type of software testing where individual units or components of a software are tested**. The purpose is to validate that each unit of the software code performs as expected. Unit Testing is done during the development (coding phase) of an application by the developers.

1. Unit tests help to fix bugs early in the development cycle and save costs.
2. It helps the developers to understand the testing code base and enables them to make changes quickly
3. Good unit tests serve as project documentation
4. Unit tests help with code re-use. Migrate both your code **and** your tests to your new project. Tweak the code until the tests run again.
5. Well-formed Unit tests will facilitate the next steps in the software testing cycle. Eg. Integrations testing, ...



Python unittest

Python unittest

Intro

The unittest unit testing framework was originally inspired by JUnit and has a similar flavor as major unit testing frameworks in other languages. It 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.

Basic example

The unittest module provides a rich set of tools for constructing and running tests. This section demonstrates that a small subset of the tools suffice to meet the needs of most users.

unittest requires that:

- You put your tests into classes as methods
- You use a series of special assertion methods in the unittest.TestCase class instead of the built-in assert statement

```
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'])
        with self.assertRaises(TypeError):
            s.split(2)
```

Example: sum() function

We're going to test the python built-in **sum()** function.

Step 1: Implement TestCase

First, we should implement a **SumTest** class extends **unittest.TestCase**

```
import unittest

class SumTest(unittest.TestCase):
    ...
```

Example: sum() function

Step 2: Writing tests

In the next step, we should create test methods seek to maximize the coverage of the sum() function.

Note: All test methods name must to starts with **test** keyword.

```
class SumTest(unittest.TestCase):

    def test1_int_list_success (self):
        self.assertEqual(sum([1, 2, 5]), 8)

    def test2_int_list_success (self):
        self.assertEqual(sum([1, -2, 10]), 9)

    def test3_float_list_success (self):
        self.assertEqual(sum([1.2, -2, 10]), 9.2)

    def test4_float_list_success (self):
        self.assertEqual(sum([1, -2, 10.1]), 9.1)
```

Python unittest

Example: sum() function

Step 2: Writing tests

In the next step, we should create test methods seek to maximize the coverage of the sum() function.

Note:

- All test methods name must starts with the **test** keyword.
- It's good to create expected failing tests to cover all aspects of the unit.
- You can add custom message to show on assertion failure.

```
class SumTest(unittest.TestCase):

    def test1_int_list_success (self):
        self.assertEqual(sum([1, 2, 5]), 8)

    def test2_int_list_success (self):
        self.assertEqual(sum([1, -2, 10]), 9
                         , "Should be 9")

    def test3_float_list_success (self):
        self.assertEqual(sum([1.2, -2, 10]), 9.2,
                         "Should be 9.2")

    def test4_float_list_success (self):
        self.assertEqual(sum([1, -2, 10.1]), 9.1
                         , "Should not be int!")

    def test5_float_list_fail (self):
        self.assertNotEqual(sum([1, -2, 10.1]), 9
                            , "Should not be int!")
```

Example: sum() function

Step 3: Running tests

Finally, we can run the test cases using **unittest.main()** or use the terminal commands:

my_test.py:

```
class SumTest(unittest.TestCase):
    ...

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

```
python -m unittest my_test
python -m unittest my_test.SumTest
python -m unittest -v my_test.SumTest
python -m unittest my_test.SumTest.test1_int_list_success
```

DON'T mention **.py**

Run specific test class

Run on verbose mode

Run on specific method

Python unittest

Example: sum() function

Result:

```
----------------------------------------
Ran 5 tests in 0.000s

OK
```

On verbose mode:

```
test1_int_list_success (my_test.SumTest) ... ok
test2_int_list_success (my_test.SumTest) ... ok
test3_float_list_success (my_test.SumTest) ... ok
test4_float_list_success (my_test.SumTest) ... ok
test5_float_list_fail (my_test.SumTest) ... ok

----------------------------------------------------
Ran 5 tests in 0.000s

OK
```

Python unittest

Some of assert methods

Full document

Method	Checks that	New in
`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`	3.1
`assertIsNot(a, b)`	`a is not b`	3.1
`assertIsNone(x)`	`x is None`	3.1
`assertIsNotNone(x)`	`x is not None`	3.1
`assertIn(a, b)`	`a in b`	3.1
`assertNotIn(a, b)`	`a not in b`	3.1
`assertIsInstance(a, b)`	`isinstance(a, b)`	3.2
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	3.2

Practice: primals

A. Write the **is_primal(n)** and **primals(n)** functions.
   - **is_primal(n: int) -> bool**
     Returns True if n is prime. (n must be greater than 1)
   - **primals(n: int) -> list**
     Returns the list of prime numbers less than n.

B. Create unit Test cases (**IsPrimalTest** & **PrimalsTest**).
   - Write at least 8 test methods for each test cases.
   - Consider the failure cases.
   - Try to maximize the test coverage by writing more and proper test methods.
   - Embed assertion method with appropriate Fail message.

Hint:
- You should utilize the **assertListEqual()** method to test results of type list.
  DO NOT USE assertEqual() (why?)