Unit-VIII
Testing in Python

By:
Mani Butwall,
Asst. Prof. (CSE)

Contents

- Testing in Python
- What is testing?
- Manual Testing and Automated Testing
- Unit test
- Writing Unit Test in Python
- Edge Cases
- Additional test cases
- Black Box vs. White Box
- Other test types, test driven development

What is 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.

Types of Testing

- White Box
- Black Box

Black Box Testing	White Box Testing
1. Black box testing techniques are also called functional testing techniques.	1. White box testing techniques are also called structural testing techniques.
2Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is NOT known to the tester	2. White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.
3. It is mainly applicable to higher levels of testing such as Acceptance Testing and System Testing	3. Mainly applicable to lower levels of testing such as Unit Testing and Integration Testing
4. Black box testing is generally done by Software Testers	4. White box testing is generally done by Software Developers
5. Programming knowledge is not required	5. Programming knowledge is required
6. Implementation knowledge is not required.	6. Implementation knowledge is required

What is the Python unittest?

- Unit testing is a technique in which particular module is tested to check by developer himself whether there are any errors. The primary focus of unit testing is test an individual unit of system to analyze, detect, and fix the errors.
- Python provides the **unittest module** to test the unit of source code. The unittest plays an essential role when we are writing the huge code, and it provides the facility to check whether the output is correct or not.
- Normally, we print the value and match it with the reference output or check the output manually.
- This process takes lots of time. To overcome this problem, Python introduces the **unittest** module. We can also check the application's performance by using it.

OOP concepts supported by unittest framework

- **Method:**
  White Box Testing method is used for Unit testing.
- **test fixture:**
  A test fixture is used as a baseline for running tests to ensure that there is a fixed environment in which tests are run so that results are repeatable.
  Examples :
  - creating temporary databases.
  - starting a server process.
- **test case:**
  A test case is a set of conditions which is used to determine whether a system under test works correctly.
- **test suite:**
  Test suite is a collection of testcases that are used to test a software program to show that it has some specified set of behaviours by executing the aggregated tests together.
- **test runner:**
  A test runner is a component which set up the execution of tests and provides the outcome to the user.

```
In [29]: import unittest
    ...:
    ...: def add(x,y):
    ...:     return x +" "+ y
    ...:
    ...:
    ...: class MyTest(unittest.TestCase):
    ...:     def test(self):
    ...:         self.assertEqual(add("Mani","Butwall"), "Mani Butwall")
    ...:
    ...:
    ...: if __name__ == '__main__':
    ...:     unittest.main()
    ...:
    ...:
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

```
In [30]: import unittest
    ...:
    ...: def add(x,y):
    ...:     return x + y
    ...:
    ...:
    ...: class MyTest(unittest.TestCase):
    ...:     def test(self):
    ...:         self.assertEqual(add("Mani","Butwall"), "Mani Butwall")
    ...:
    ...:
    ...: if __name__ == '__main__':
    ...:     unittest.main()
    ...:
    ...:
F
```

```
....
F
======================================================================
FAIL: test (__main__.MyTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-30-ce8b66e4edaf>", line 9, in test
    self.assertEqual(add("Mani","Butwall"), "Mani Butwall")
AssertionError: 'ManiButwall' != 'Mani Butwall'
- ManiButwall
+ Mani Butwall
?     +


----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

```
In [31]: import unittest
    ...:
    ...: def add(x,y):
    ...:     return x + y
    ...:
    ...:
    ...: class MyTest(unittest.TestCase):
    ...:     def test(self):
    ...:         self.assertEqual(add(3,4), 7)
    ...:
    ...:
    ...: if __name__ == '__main__':
    ...:     unittest.main()
    ...:
    ...:
.
----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

```
In [32]: import unittest
    ...:
    ...: def add(x,y):
    ...:     return x + y
    ...:
    ...:
    ...: class MyTest(unittest.TestCase):
    ...:     def test(self):
    ...:         self.assertEqual(add(3,4), 8)
    ...:
    ...:
    ...: if __name__ == '__main__':
    ...:     unittest.main()
    ...:
    ...:
```

```
    ....
F
======================================================================
FAIL: test (__main__.MyTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-32-199cdcbd2aaf>", line 9, in test
    self.assertEqual(add(3,4), 8)
AssertionError: 7 != 8

----------------------------------------------------------------------
Ran 1 test in 0.002s

FAILED (failures=1)
```

```
In [35]: def test_sum2():
    ...:     assert sum([2, 3, 5]) == 10, "It should be 10"
    ...:
    ...:
    ...: def test_sum_tuple():
    ...:     assert sum((1, 3, 5)) == 10, "It should be 10"
    ...:
    ...:
    ...: if __name__ == "__main__":
    ...:     test_sum2()
    ...:     test_sum_tuple()
    ...:     print("Everything is correct")
    ...:
    ...:
Traceback (most recent call last):

  File "<ipython-input-35-42dcafb05feb>", line 11, in <module>
    test_sum_tuple()

  File "<ipython-input-35-42dcafb05feb>", line 6, in test_sum_tuple
    assert sum((1, 3, 5)) == 10, "It should be 10"

AssertionError: It should be 10
```

```
In [11]: import unittest
    ...:
    ...: class MyTest(unittest.TestCase):
    ...:
    ...:     def test_true(self):
    ...:         self.assertTrue('FOo'.isupper())
    ...:
    ...:     def test_in(self):
    ...:         self.assertIn('5', ['2', 7, 'a', '1'])
    ...:
    ...:
    ...: if __name__ == '__main__':
    ...:     unittest.main()
    ...:
    ...:
```

```
    ....
FF
======================================================================
FAIL: test_in (__main__.MyTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-11-b2fa6d819d9b>", line 9, in test_in
    self.assertIn('5', ['2', 7, 'a', '1'])
AssertionError: '5' not found in ['2', 7, 'a', '1']

======================================================================
FAIL: test_true (__main__.MyTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "<ipython-input-11-b2fa6d819d9b>", line 6, in test_true
    self.assertTrue('FOo'.isupper())
AssertionError: False is not true

----------------------------------------------------------------------
Ran 2 tests in 0.002s

FAILED (failures=2)
```

```
In [12]: import unittest
    ...:
    ...: class MyTest(unittest.TestCase):
    ...:
    ...:     def test_true(self):
    ...:         self.assertTrue('FOO'.isupper())
    ...:
    ...:     def test_in(self):
    ...:         self.assertIn('1', ['2', 7, 'a', '1'])
    ...:
    ...:
    ...: if __name__ == '__main__':
    ...:     unittest.main()
    ...:
    ...:
..
----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
```

Testing the Code

- We can test our code using many ways.
- **Automate vs. Manual Testing**
- Manual testing has another form, which is known as exploratory testing. It is a testing which is done without any plan. To do the manual testing, we need to prepare a list of the application; we enter the different inputs and wait for the expected output.
- Every time we give the inputs or change the code, we need to go through every single feature of the list and check it.
- It is the most common way of testing and it is also time-consuming process.
- On the other hand, the automated testing executes the code according to our code plan which means it runs a part of the code that we want to test, the order in which we want to test them by a script instead of a human.
- Python offers a set of tools and libraries which help us to create automated tests for the application.

Unit Tests vs. Integration Tests

- Suppose we want to check the lights of the car and how we might test them. We would turn on the light and go outside the car or ask the friend that lights are on or not. The turning on the light will **consider** as the test step, and go outside or ask to the friend will know as the **test assertion.** In the integration testing, we can test multiple components at once.
- These components can be anything in our code, such as functions, classes and module that we have written.
- But there is a limitation of the integration testing; what if an integration test doesn't give the expected result. In this situation, it will be very hard to recognize which part of the system is falling. Let's take the previous example; if the light didn't turn on, the battery might be dead, blub is broken, car's computer have failed.
- That's why we consider unit testing to get to know the exact problem in the tested code.
- Unit testing is a smaller test, it checks a single component that it is working in right way or not. Using the unit test, we can separate what necessities to be fixed in our system.

- We have seen the two types of testing so far; an integration test checks the multiple components; where unit test checks small component in or application.
- Let's understand the following example.
- We apply the unit testing Python built-in function **sum()** against the known output. We check that the sum() of the number **(2, 3, 5)** equals 10.

unittest

- The unittest is built into the Python standard library since 2.1. The best thing about the unittest, it comes with both a test framework and a test runner. There are few requirements of the unittest to write and execute the code.
- The code must be written using the classes and functions.
- The sequence of distinct assertion methods in the **TestCase** class apart from the built-in asserts statements.
- Let's implement the above example using the unittest case.

Method	Description
.assertEqual(a, b)	a == b
.assertTrue(x)	bool(x) is True
.assertFalse(x)	bool(x) is False
.assertIs(a, b)	a is b
.assertIsNone(x)	x is None
.assertIn(a, b)	a in b
.assertIsInstance(a, b)	isinstance(a, b)
.assertNotIn(a, b)	a not in b
.assertNotIsInstance(a,b)	not isinstance(a, b)
.assertIsNot(a, b)	a is not b

Advantages of using Python Unit testing

1. It helps you to detect bugs early in the development cycle
2. It helps you to write better programs
3. It syncs easily with other testing methods and tools
4. It will have many fewer bugs
5. It is easier to modify in future with very less consequence


Thank
You