Python Unit Testing

In this tutorial, we will implement unit testing using the Python. Unit testing using Python is a huge topic itself, but we will cover a few essential concepts.

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.

We will learn how to create a basic test, finds the bugs, and execute it before the code delivers to the users.

Testing the Code

We can test our code using many ways. In this section, we will learn the basic steps towards advanced methods.

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.

```
assert sum([ 2, 3, 5]) == 10, "Should be 10"
```

Above line will return the right result because values are correct. If we pass the wrong arguments it will return the **Assertion error**. For example -

```
assert sum([1, 3, 5]) == 10, "Should be 10"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: Should be 10
```

We can put the above code into the file and execute it again at the command line.

```
def test_sum():
    assert sum([2, 3, 5]) == 10, "It should be 10"

if __name__ == "__main__":
    test_sum()
    print("Everything passed")
```

Output:

```
$ python sum.py
Everything is correct
```

In the following example, we will pass the tuple for testing purpose. Create a new file named **test_sum2.py.**

Example - 2:

```
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()
```



```
g is correct")
```

Output:

```
Everything is correct
Traceback (most recent call last):
  File "<string>", line 13, in <module>
File "<string>", line 9, in test_sum_tuple
AssertionError: It should be 10
```

Explanation -

In the above code, we have passed the wrong input to the **test_sum_tuple().** The output is dissimilar to the predicted result.

The above method is good but what if there are multiple errors. Python interpreter would give an error immediately if the first error is encountered. To remove this problem, we use the test runners.

> “ Test runner applications specially designed for testing the output, running test and give tools for fixing and diagnosing tests and applications. ”

Choosing a Test Runner

Python contains many test runners. The most popular build-in Python library is called **unittest.** The unittest is portable to the other frameworks. Consider the following three top most test runners.

- unittest
- nose Or nose2
- pytest

We can choose any of them according to our requirements. Let's have a brief introduction.

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.

Example -

```
import unittest
class TestingSum(unittest.TestCase):

    def test_sum(self):
        self.assertEqual(sum([2, 3, 5]), 10, "It should be 10")
    def test_sum_tuple(self):
        self.assertEqual(sum((1, 3, 5)), 10, "It should be 10")
```

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

Output:

```
.F
-
FAIL: test_sum_tuple (__main__.TestingSum)
--
Traceback (most recent call last):
  File "<string>", line 11, in test_sum_tuple
AssertionError: 9 != 10 : It should be 10

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
Traceback (most recent call last):
  File "<string>", line 14, in <module>
  File "/usr/lib/python3.8/unittest/main.py", line 101, in __init__
    self.runTests()
  File "/usr/lib/python3.8/unittest/main.py", line 273, in runTests
    sys.exit(not self.result.wasSuccessful())
SystemExit: True
```

As we can see in the output, it shows the **dot(.)** for the successful execution and **F** for the one failure.

nose

Sometimes, we need to write hundreds or thousands of test lines for application; it becomes so difficult to understand.

The nose test runner can be a suitable replacement of the unittest test runners because it is compatible with any tests writing using the unittest framework. There are two types of nose - nose and nose2. We recommend using nose2 because it is a latest version.

Working with the nose2, we need to install it using the following command.

```
pip install nose2
```

Run the following command in the terminal to test the code using nose2.

```
python -m nose2
```

The output is as follows.

```
FAIL: test_sum_tuple (__main__.TestSum)
--
                ecent call last):
```

```
  File "test_sum_unittest.py", line 10, in test_sum_tuple
    self.assertEqual(sum((2, 3, 5)), 10, "It should be 10")
AssertionError: It should be 10

--
Ran 2 tests in 0.001s

FAILED (failures=1)
```

The nose2 provides many command line flags for filtering the test. You can learn more from its official documentation.

pytest

The pytest test runner supports the execution of **unittest** test cases. The actual benefit of the **pytest** is to writing **pytest** test cases. The pytest test cases are generally sequence of methods in the Python file starting.

The pytest provides the following benefits -

- It supports the built-in assert statement instead of using a special **assert*()** methods.
- It also provides support for cleaning for test cases.
- It can rerun from the last cases.
- It has an ecosystem of hundreds of plugin to extend the functionality.

Let's understand the following example.

Example -

```
def test_sum():
    assert sum([2, 3, 5]) == 10, "It should be 10"
def test_sum_tuple():
    assert sum((1, 2, 5)) == 10, "It should be 10"
```

Writing the First Test

Here we will apply all the concepts that we have learned in earlier section. First, we need to create a file name **test.py** or anything. Then make inputs and execute the code being tested, capturing the output. After successfully run the code, match the output with an expected result.

First, we create the file **my_sum** file and write code in it.

```
def sum(arg):
    total = 0
    for val in arg:
        total += val
    return total
```

We initialized the total variable which iterates over all the values in arg.

name **test.py** with the following code.

Example -

```
import unittest

from my_sum import sum


class CheckSum(unittest.TestCase):
    def test_list_int(self):

        data = [1, 2, 3]
        result = sum(data)
        self.assertEqual(result, 6)

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

Output:

```
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

Explanation:

In the above code, we imported **sum()** from the **my_sum package** that we created. We have defined the **Checkclass,** which inherits from **unittest.TestCase.** There is a test methods - **.test_list_int(),** to test the integer.

After running the code, it returns **dot(.)** which means there is no error in the code.

Let's understand another example.

Example - 2

```
class Person:
    name1 = []

    def set_name(self, user_name):
        self.name1.append(user_name)
        return len(self.name1) - 1

    def get_name(self, user_id):
        if user_id >= len(self.name1):
            return ' No such user Find'
        else:
            return self.name1[user_id]
```

```
if __name__ == '__main__':
    person = Person()
    print('Peter Decosta has been added with id ', person.set_name('Peter'))
    print('The user associated with id 0 is ', person.get_name(0))
```

Output:

```
Peter Decosta has been added with id 0
The user associated with id 0 is Peter
```

Python Basic Functions and Unit Test Output

The unittest module produces three possible outcomes. Below are the potential outcomes.

1. **OK -** If all tests are passed, it will return OK.
2. **Failure -** It will raise an **AssertionError** exception, if any of tests is failed.
3. **Error -** If any errors occur instead of Assertion error.

Let's see the following basic functions.

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

Python Unit Test Example

```
import unittest

# First we import the class which we want to test.
PerClass
```

```
class Test(unittest.TestCase):
    """
    The basic class that inherits unittest.TestCase
    """
    person = PerClass.Person()  # instantiate the Person Class
    user_id = []  # This variable stores the obtained user_id
    user_name = []  # This variable stores the person name

    # It is a test case function to check the Person.set_name function
    def test_0_set_name(self):
        print("Start set_name test\n")

        for i in range(4):
            # initialize a name
            name = 'name' + str(i)
            # put the name into the list variable
            self.user_name.append(name)
            # extraxt the user id obtained from the function
            user_id = self.person.set_name(name)
            # check if the obtained user id is null or not
            self.assertIsNotNone(user_id)
            # store the user id to the list
            self.user_id.append(user_id)
        print("The length of user_id is = ", len(self.user_id))
        print(self.user_id)
        print("The length of user_name is = ", len(self.user_name))
        print(self.user_name)
        print("\nFinish set_name test\n")

    # Second test case function to check the Person.get_name function
    def test_1_get_name(self):
        print("\nStart get_name test\n")

        # total number of stored user information
        length = len(self.user_id)
        print("The length of user_id is = ", length)
        print("The lenght of user_name is = ", len(self.user_name))
        for i in range(6):
            # if i not exceed total length then verify the returned name
            if i < length:
                # if the two name not matches it will fail the test case
                self.assertEqual(self.user_name[i], self.person.get_name(self.user_id[i]))
            else:
                print("Testing for get_name no user test")
                # exceeds then check the 'no such user' type message
```

```
            self.assertEqual('There is no such user', self.person.get_name(i))
        print("\nFinish get_name test\n")


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

Output:

```
Start set_name test

The length of user_id is =  4
[0, 1, 2, 3]
The length of user_name is =  4
['name0', 'name1', 'name2', 'name3']

Finish set_name test


Start get_name test

The length of user_id is =  4
The lenght of user_name is =  4
Testing for get_name no user test
.F
======================================================================
FAIL: test_1_get_name (__main__.Test)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/multiprocessing.py", line 502, in test_1_get_name
    self.assertEqual('There is no such user', self.person.get_name(i))
AssertionError: 'There is no such user' != ' No such user Find'
- There is no such user
+  No such user Find


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

FAILED (failures=1)
```

Advance Testing Scenario

We must follow the given step while creating test for the application.

ssary input

de, taking the output.

- Match the output with an expected result.

Creating inputs such as static value for the input like a string or numbers is a slightly complex task. Sometimes, we need to create an instance of a class or a context.

The input data that we create is known as a fixture. We can reuse fixtures in our application.

When we run the code repeatedly and pass the different values each time and expecting the same result, this process is known as **parameterization.**

Handling Expected Failures

In the earlier example, we pass the integer number to test **sum(); what** happens if we pass the bad value, such as a single integer or a string?

The **sum()** will throw an error as expected. It would happen due to failed test.

We can use the **.assertRaises()** to handle the expected errors. It is used inside **with** statement. Let's understand the following example.

Example -

```
import unittest
from my_sum import sum

class CheckSum(unittest.TestCase):
    def test_list_int(self):

        #  Test that it can sum a list of integers

        data = [1, 2, 3]
        res = sum(data)
        self.assertEqual(res, 6)

    def test_bad_type(self):
        data = "Apple"
        with self.assertRaises(TypeError):
            res = sum(data)

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

Output:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.006s

OK
```

Python unittest Skip Test

We can skip an individual test method or **TestCase** using the skip test technique. The fail will not count as a failure in TestResult.

Consider the following example to skip the method unconditionally.

Example -

```
import unittest

def add(x,y):
    c = x + y
    return c

class SimpleTest(unittest.TestCase):
  @unittest.skip("The example skipping method")
  def testadd1(self):
    self.assertEquals(add(10,5),7)

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

Output:

```
s
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK (skipped=1)
```

Explanation:

In the above example, the **skip()** method prefixed by the @token. It takes the one argument a log message where we can describe the reason for skip. The **s** character denotes that a test has been successfully skipped.

We can skip a particular method or block based on the specific condition.

Example - 2:

```
import unittest



class suiteTest(unittest.TestCase):
   a = 100
   b = 40


elf.b
```

```
        self.assertEqual(res, 100)

    @unittest.skipIf(a > b, "Skip because a is greater than b")
    def test_sub(self):

        res = self.a - self.b
        self.assertTrue(res == -10)

    @unittest.skipUnless(b == 0, "Skip because b is eqaul to zero")
    def test_div(self):

        res = self.a / self.b
        self.assertTrue(res == 1)

    @unittest.expectedFailure
    def test_mul(self):

        res = self.a * self.b
        self.assertEqual(res == 0)


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

Output:

```
Fsx.
======================================================================
FAIL: test_add (__main__.suiteTest)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:/Users/DEVANSH SHARMA/PycharmProjects/Hello/multiprocessing.py", line 539, in test_add
    self.assertEqual(res, 100)
AssertionError: 50 != 100

----------------------------------------------------------------------
Ran 4 tests in 0.001s

FAILED (failures=1, skipped=1, expected failures=1)
```

Explanation:

As we can see in the output, the conditions b == 0 and a>b is true so the **test_mul()** method has skipped. On the other hand, **test_mul** has been marked as an expected failure.

Conclusion

We have discussed the all-important concept related to Python unit testing. As a beginner, we need to write the smart, maintainable methods to validate our code. Once we get a decent command over the Python unit test, we can switch to other frameworks such as **the pytest** and leverage more advanced features.

← Prev

Next →

Feedback

- Send your Feedback to feedback@javatpoint.com

Help Others, Please Share