

UNIT TESTING

Chapter - 17

WHAT IS UNIT TESTING?



Unit testing is a software testing method by which individual units of source code are put under various tests to determine whether they are fit for use.



It determines and ascertains the quality of your code.



The developers are expected to write automated test scripts, which ensures that each and every section or a unit meets its design and behaves as expected.

The unittest module

- Though writing manual tests for your code is definitely a tedious and time-consuming task, Python's built-in unit testing framework has made life a lot easier.
- The unit test framework in Python is called `unittest`, which comes packaged with Python.
- Unit testing makes your code future proof since you anticipate the cases where your code could potentially fail or produce a bug. Though you cannot predict all of the cases, you still address most of them.

Unit testing

- A unit could be bucketed into various categories:
 - An entire module,
 - An individual function,
 - A complete interface like a class or a method.
- The best ways to write unit tests for your code is to first start with a smallest testable unit your code could possibly have, then move on to other units and see how that smallest unit interacts with other units, this way you could build up a comprehensive unit test for your applications.

Test case and Test suite

Test case

- A test case is the individual unit of testing. It checks for a specific response to a particular set of inputs.
- `unittest` provides a base class, `TestCase`, which may be used to create new test cases.
- A testcase is created by subclassing `unittest.TestCase`.

Test suite

- A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

Writing Unit Tests

- One of the most important classes provided by the `unittest` module is named `TestCase`.
- The `TestCase` class provides several assert methods to check for and report failures. The table on the right lists the most commonly used methods.

Method	Checks that	New in
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assert IsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1

```
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
```

➊ cuboid_volume.py > ⚙ find_cuboid_volume

```
1 ✓ def find_cuboid_volume(l):
2     |
3         return (l*l*l)
4
5
6 def find_cuboid_volume(l):
7     return (l*l*l)
8
9 length = [2,1.1, -2.5,True, 2j, 'two']
10
11 for i in range(len(length)):
12     print ("The volume of cuboid:",find_cuboid_volume(length[i]))
```

PROB

PS C:\

The volume of cuboid: 8

The volume of cuboid: 1.331000000000004

The volume of cuboid: -15.625

The volume of cuboid: 1

The volume of cuboid: (-0-8i)

Traceback (most recent call last):

File "c:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine\cuboid volume.py", line 9, in <module>

```
print ("The volume of cuboid:",find_cuboid_volume(length[i]))
```

```
File "c:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine\cuboid_volume.py", line 3, in find_cuboid_volume
    return (l*l*l)
```

```
TypeError: can't multiply sequence by non-int of type 'str'
```

Writing Unit Tests

- Unit tests are usually written as a separate code in a different file, and there could be different naming conventions that you could follow.
- You could either write the name of the unit test file as the name of the code/unit + test separated by an underscore or test + name of the code/unit separated by an underscore.
- For example, let's say the above code file name is `cuboid_volume.py`, then your unit test code name could be `cuboid_volume_test.py` or `test_volume_cuboid.py`

Writing Unit Tests

- First, let's create a python file with the name `test_volume_cuboid.py`, which will have the unit testing code.
- Then import the `unittest` module
- Then create a class `TestCuboid` that inherits the `unittest` module.
- Then you would define various methods that you would want your unit test should check with your function `cuboid_volume`.

Writing Unit Tests

- The first function you will define is `test_volume`, which will check whether the output your `cuboid_volume` gives is equal to what you expect.
- To achieve this, you will make use of the `assertAlmostEqual` method.

test_cuboid_volume.py > TestCuboid > test_volume

```
1 from cuboid_volume import *
2 import unittest
3
4 class TestCuboid(unittest.TestCase):
5     def test_volume(self):
6         self.assertAlmostEqual(find_cuboid_volume(2),8)
7         self.assertAlmostEqual(find_cuboid_volume(1),1)
8         self.assertAlmostEqual(find_cuboid_volume(0),0)
9
10        self.assertAlmostEqual(find_cuboid_volume(5.5),166.375)
11
```

Run the tests

- Let's run the script. You would run the unittest module as a script by specifying -m while running it.
- We invoked the Python library module named unittest with python -m unittest. Then, we provided the path to our file as an argument.

```
python -m unittest test_cuboid_volume.py
```

cubeoid_volume.py > ...

```
1 def find_cuboid_volume(l):
2     return (l*l*l)
3
4
5 #length = [2,1.1, -2.5,True, 2j, 'two']
6 #for i in range(len(length)):
7 #print ("The volume of cuboid:",find_cuboid_volume(length[i]))
8
9 #    print ("The volume of cuboid. ,find_cuboid_volume(length[i]))
```

```
1 from cuboid_volume import *
2 import unittest
3
4
5 class TestCuboid(unittest.TestCase):
6     def test_volume(self):
7         self.assertAlmostEqual(find_cuboid_volume(2),8)
8         self.assertAlmostEqual(find_cuboid_volume(1),1)
9         self.assertAlmostEqual(find_cuboid_volume(0),0)
10        self.assertAlmostEqual(find_cuboid_volume(5.5),166.375)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

PS C:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine> python -m unittest test_cuboid_volume.py

.

-

Ran 1 test in 0.000s

OK

```
py test_cuboid_volume.py > ...
4 class TestCuboid(unittest.TestCase):
5     def test_volume(self):
6         self.assertAlmostEqual(find_cuboid_volume(2),8)
7         self.assertAlmostEqual(find_cuboid_volume(1),1)
8         self.assertAlmostEqual(find_cuboid_volume(0),0)
9         self.assertAlmostEqual(find_cuboid_volume(5.5),166.375)
10
11 if __name__ == '__main__':
12     unittest.main()
13
14
```

- The last block of code allows us to run all the tests just by running the file.
- main associated with unittest is actually an instance of TestCuboid which, when instantiated, runs all your tests.
- The constructor also takes your current file as the default module containing the tests (module='__main__').

Unit test

- The test ran successfully and returned, OK, which means the cuboid_volume function works as you would expect it too. The single . on the first line of the output represents our passed test.
- Let's see what happens when one of the assertAlmostEqual methods fails.
- Notice that the last assert statement has been modified.

```
self.assertAlmostEqual(find_cuboid_volume(0),0)
self.assertAlmostEqual(find_cuboid_volume(5.5),0)
```

```
PS C:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine> python -m unittest test_cuboid_volume.py
F
=====
FAIL: test_volume (test_cuboid_volume.TestCuboid)
-----
Traceback (most recent call last):
  File "C:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine\test_cuboid_volume.py", line 10, in test_volume
    self.assertAlmostEqual(find_cuboid_volume(5.5),0)
AssertionError: 166.375 != 0 within 7 places (166.375 difference)

-----
Ran 1 test in 0.001s

FAILED (failures=1)
```

- Well, from the above output, you can observe that the last assert statement resulted in an AssertionError, hence a unit test failure.
- F is the output when unittest runs a test that fails.
- Python's unit test module shows you the reason for the failure, along with the number of failures your code has.

- Now let's explore another assert method, i.e., `assertRaises`, which will help you in finding out whether your function `cuboid_volume` handles the input values correctly.
- Let's say you want to test whether your function `cuboid_volume` handles the type of input, for example, if you pass a boolean as an input will it handle that input either as an exception or with an if condition since the length of the cuboid can never be a True/False.

```
test_cuboid_volume.py > ...
```

```
5     class TestCuboid(unittest.TestCase):
6
7         def test_volume(self):
8             self.assertAlmostEqual(find_cuboid_volume(2),8)
9             self.assertAlmostEqual(find_cuboid_volume(1),1)
10            self.assertAlmostEqual(find_cuboid_volume(0),0)
11            self.assertAlmostEqual(find_cuboid_volume(5.5),166.375)
12
13    def test_input_value(self):
14        self.assertRaises(TypeError,find_cuboid_volume,True)
15
16
17    if __name__ == '__main__':
18        unittest.main()
```

/Python

```
Traceback (most recent call last):
  File "c:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine\test_cuboid_volume.py", line 13, in test_i
nput_value
    self.assertRaises(TypeError,find_cuboid_volume,True)
AssertionError: TypeError not raised by find_cuboid_volume
```

```
Ran 2 tests in 0.001s
```

```
FAILED (failures=1)
```

- So, from the above output, it is evident that your code `cuboid_volume.py` doesn't take care of the input being passed to it properly.
- Let's add a condition in the `cuboid_volume.py` to check whether the input or length of the cuboid is a boolean or a string and raise an error.

⚡ cuboid_volume.py > ⚑ find_cuboid_volume

```
● 1 ✓ def find_cuboid_volume(l):
    if type(l) not in [int,float]:
        find_cuboid_volume(l)
            if type(l) not in [int,float]:
                raise TypeError('The length of cuboid can only be an integer or float')
            return (l*l*l)

    # # length = [2,1.1, -2.5, 2j, 'two']

    # for i in range(len(length)):
    #     print ("The volume of cuboid:",find_cuboid_volume(length[i]))
```

```
➊ test_cuboid_volume.py > ...
```

```
1  from cuboid_volume import *
2  import unittest
3
4
5  class TestCuboid(un
6      def test_volume
7          self.assert
8          self.assert
9          self.assert
10         self.assert
11
12     def test_input_
13         # Parameter
14         # 1: Type o
15         # 2: Volume
16         # 3: The da
17         self.assert
18
19     if __name__ == '__m
20         unittest.main()
21
22
```

```
PROBLEMS OUTPUT DEBUG CO
```

```
from cuboid volume import *
import unittest

class TestCuboid(unittest.TestCase):
    def test volume(self):
        self.assertAlmostEqual(find_cuboid_volume(2),8)
        self.assertAlmostEqual(find_cuboid_volume(1),1)
        self.assertAlmostEqual(find_cuboid_volume(0),0)
        self.assertAlmostEqual(find_cuboid_volume(5.5),166.375)

    def test input value_type(self):
        # Parameters of assertRaises:
        # 1: Type of error # 2: Volume function
        # 3: The dataType which should raise an error
        self.assertRaises(TypeError,find_cuboid_volume,complex)

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

```
.exe c:/Users/Angela/Desktop/PythonScripts/Batch-5/Mine/test_cuboid_volume.py
```

```
..
```

```
Ran 2 tests in 0.000s
```

```
OK
```

Adding more tests

- Now let's add some more tests that will prevent any other datatypes like strings, Boolean and a negative number as input while computing the volume of the cuboid.
- We need to use assertRaises to validate the arguments passed

```
1  from cuboid volume import *
2  import unittest
3
4  class TestCuboid(unittest.TestCase):
5      def test volume(self):
6          self.assertAlmostEqual(find_cuboid_volume(2),8)
7          self.assertAlmostEqual(find_cuboid_volume(1),1)
8          self.assertAlmostEqual(find_cuboid_volume(0),0)
9          self.assertAlmostEqual(find_cuboid_volume(5.5),166.375)
10
11     def test input value_type(self):
12         # Parameters of assertRaises:
13         # 1: Type of error # 2: Volume function
14         # 3: The dataType which should raise an error
15         self.assertRaises(TypeError,find_cuboid_volume,complex)
16         self.assertRaises(TypeError,find_cuboid_volume,bool)
17         self.assertRaises(TypeError,find_cuboid_volume,str)
18
19     def test input value negative(self):
20         # Parameters of assertRaises:
21         # 1: Type of error
22         # 2: Volume function
23         # 3: The values which should raise the error.
24         self.assertRaises(ValueError,find_cuboid_volume,-5)
25
26
27     if __name__ == '__main__':
28         unittest.main()
29
30     unittest.main()
```

Result

```
.exe c:/Users/Angela/Desktop/PythonScripts/Batch-5/Mine/test_cuboid_volume.py
...
-----
Ran 3 tests in 0.001s
OK
```

Solving the problems

In our function, we mainly encountered two major problems:

- `cuboid_volume(length)` gave output on negative values. In such a situation, we call the `assertRaises` method, which returns a `ValueError` if the length is negative. For testing purposes, we have given the length as-5.
- In `cuboid_volume.py`, we define an if condition to only calculate the volume if the radius is positive. If the radius is not positive, a value error will be raised. Now, our code is aligned with our third test case.

Solving the problems

- `cuboid_volume(length)` gave output on Boolean, complex and string values. In such a situation, we call the `assertRaises` method, which returns `TypeError` if the radius is boolean or string (for testing purposes, supplied Boolean,complex and string values). Now we know that for Boolean,complex and string inputs, the `cuboid_volume(length)` function will raise an error.
- So, in `cuboid_volume(length)` , we define an if condition to only calculate the volume if the radius data type is float or int. If not, a type error will be raised. Our code is now also aligned with our second test case.

cubeoid_volume.py > ...

```
1     def find_cuboid_volume(l):
2         if type(l) not in [int,float]:
3             raise TypeError('The length of cuboid can only be an integer or float')
4         elif l < 0:
5             raise ValueError(" The length of the cuboid can be only a positive number")
6         return (l*l*l)
7
8
9     length = [2,1.1, -2.5,True, 2j, 'two']
10
11    for i in range(len(length)):
12        print ("The volume of cuboid:",find_cuboid_volume(length[i]))
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

[>] Python + ▾ []

```
PS C:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine> & C:/Users/Angela/AppData/Local/Programs/Python/Python.exe c:/Users/Angela/Desktop/PythonScripts/Batch-5/Mine/cuboid_volume.py
The volume of cuboid: 8
The volume of cuboid: 1.3310000000000004
Traceback (most recent call last):
  File "c:/Users/Angela/Desktop/PythonScripts/Batch-5\Mine\cuboid_volume.py", line 12, in <module>
    print ("The volume of cuboid:",find_cuboid_volume(length[i]))
  File "c:/Users/Angela/Desktop/PythonScripts/Batch-5\Mine\cuboid_volume.py", line 5, in find_cuboid_volume
    raise ValueError(" The length of the cuboid can be only a positive number")
ValueError: The length of the cuboid can be only a positive number
```

- When we run the code above, it gives an error for boolean and negative and string values.
- This error tells us that our test has failed, which means we are on the right track.
- So, we will remove the invalid values. Now, our test gives the correct output.

cuboid_volume.py > ...

```
1 def find_cuboid_volume(l):
2     if type(l) not in [int,float]:
3         raise TypeError('The length of cuboid can only be an integer or float')
4     elif l < 0:
5         raise ValueError(" The length of the cuboid can be only a positive number")
6     return (l*l*l)
7
8 length = [2,1.1]
9
10 for i in range(len(length)):
11     print("The volume of cuboid:",find_cuboid_volume(length[i]))
12
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine> & C:/Users/Angela/AppData/Local/Programs
.exe c:/Users/Angela/Desktop/PythonScripts/Batch-5/Mine/cuboid_volume.py
The volume of cuboid: 8
The volume of cuboid: 1.3310000000000004
PS C:\Users\Angela\Desktop\PythonScripts\Batch-5\Mine>
```

More Examples

```
test_string_methods.py
1 import unittest
2
3 class TestStringMethods(unittest.TestCase):
4
5     def test_upper(self):
6         self.assertEqual('foo'.upper(), 'FOO')
7
8     def test_isupper(self):
9         self.assertTrue('FOO'.isupper())
10        self.assertFalse('Foo'.isupper())
11
12     def test_split(self):
13         s = 'hello world'
14         self.assertEqual(s.split(), ['hello', 'world'])
15
16         if __name__ == '__main__':
17             unittest.main()
```

```
python/Python39/python.exe c:/Users/Angela/Desktop/PythonScripts/Batch-5/Mine/test_string_methods.py
```

```
...
```

```
Ran 3 tests in 0.001s
```

```
OK
```

Assert methods

- There are a lot of assert methods in the unit test module of Python, which could be leveraged for your testing purposes, as shown in the previous slides.
- To know in detail about the assert methods, check out Python's official documentation. -

<https://docs.python.org/3/library/unittest.html>