# Python unittest



**Summary**: in this tutorial, you'll learn about the unit test concept and how to use the Python `unittest` module to perform unit testing.

## What is a unit test

A unit test is an automated test that:

- Verifies a small piece of code called a unit. A unit can be a function (https://www.pythontutorial.net/python-basics/python-functions/) or a method of a class (https://www.pythontutorial.net/python-oop/python-class/) .

- Runs very fast.
- Executes in an isolated manner.

The idea of unit testing is to check each small piece of your program to ensure it works properly. It's different from integration testing which tests that different parts of the program work well together.

The goal of a unit test is to find bugs. Also, a unit test can help refactor existing code to make it more testable and robust.

Python provides you with a built-in module `unittest` that allows you to carry out unit testing effectively.

## xUnit terminology

The `unittest` module follows the xUnit philosophy. It has the following major components:

- **System under test** is a function, a class, a method that will be tested.
- **Test case class** ( `unittest.TestCase` ): is the base class for all the test classes. In other words, all test classes are subclasses of the TestCase class in the `unittest` module.
- **Test fixtures** (https://www.pythontutorial.net/python-unit-testing/python-test-fixtures/) are methods that execute before and after a test method executes.
- **Assertions** are methods that check the behavior of the component being tested.
- **Test suite** is a group of related tests executed together.
- **Test runner** is a program that runs the test suite.

## Python unittest example

Suppose you have `Square` class that has a property called `length` and a method `area()` that returns the area of the square. The `Square` class is in the `square.py` module:

```
class Square:
    def __init__(self, length) -> None:
        self.length = length

    def area(self):
        return self.length * self.length
```

To test the `Square` class, you create a new file called `test_square.py` file and import the `unittest` module like this:

```
import unittest
```

Since the `test_square.py` needs to access the `Square` class, you have to import it from the `square.py` module:

```
import unittest

from square import Square
```

To create a test case, you define a new class called `TestSquare` that inherits from the `TestCase` class of the `unittest` module:

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

```
    pass
```

To test the `area()` method, you add a method called `test_area()` to the `TestSquare` class like this:

```
import unittest

from square import Square


class TestSquare(unittest.TestCase):
    def test_area(self):
        square = Square(10)
        area = square.area()
        self.assertEqual(area, 100)
```

In the `test_area()` method:

- First, create a new instance of the `Square` class and initialize its radius with the number 10.
- Second, call the `area()` method that returns the area of the square.
- Third, call the `assertEqual()` method to check if the result returned by the `area()` method is equal to an expected area ( `100` ).

If the area is equal to 100, the `assertEqual()` will pass the test. Otherwise, the `assertEqual()` will fail the test.

Before running the test, you need to call the `main()` function of the `unittest` module as follows:

```
import unittest

from square import Square


class TestSquare(unittest.TestCase):
    def test_area(self):
        square = Square(10)
        area = square.area()
        self.assertEqual(area, 100)


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

To run the test, you open the terminal, navigate to the folder, and execute the following command:

```
python test_square.py
```

If you use Linux or macOS, you need to use the python3 command instead:

```
python3 test_square.py
```

It'll output the following:

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

OK
```

The output indicates that one test has passed denoted by the dot (.) If a test failed, you would see the letter `F` instead of the dot (.)

To get more detailed information on the test result, you pass the `verbosity` argument with the value 2 to the `unittest.main()` function:

```
import unittest

from square import Square


class TestSquare(unittest.TestCase):
    def test_area(self):
        square = Square(10)
        area = square.area()
        self.assertEqual(area, 100)
```

```
if __name__ == '__main__':
    unittest.main(verbosity=2)
```

If you run the test again:

```
python test_square.py
```

you'll get the detailed information:

```
test_area (__main__.TestSquare) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.001s

OK
```

The output list the test case with the result ok this time instead of the dot (.)

## Running tests without calling unittest.main() function

First, remove the `if` block that calls the `unittest.main()` function:

```
import unittest
```

```
from square import Square



class TestSquare(unittest.TestCase):
    def test_area(self):
        square = Square(10)
        area = square.area()
        self.assertEqual(area, 100)
```

Second, execute the following command to run the test:

```
python3 -m unittest
```

This command discovers all the test classes whose names start with `Test*` located in the `test_*` file and execute the test methods that start with `test*`. the `-m` option stands for the module.

In this example, the command executes the `test_area()` method of the `TestSquare` class in the `test_square.py` test module.

If you use macOS or Linux, you need to use the `python3` command instead:

```
python3 -m unittest
```

It'll return something like:

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

OK
```

To display more information, you can add `-v` option to the above command. `v` stands for verbosity. It's like calling the unittest. `main()` with verbosity argument with value 2.

```
python -m unittest -v
```

Output:

```
test_area (test_square.TestSquare) ... ok



----------------------------------------------------------------------
Ran 1 tests in 0.000s

OK
```

## Testing expected exceptions

The `Square` constructor accepts a `length` parameter. The `length` parameter should be either an `int` or `float`. If you pass the value that is not in these types, the `Square` constructor should raise (https://www.pythontutorial.net/python-oop/python-raise-exception/) a `TypeError` exception.

To test if the `Square` constructor raises the `TypeError` exception, you use the `assertRaises()` method in a context manager (https://www.pythontutorial.net/advanced-python/python-context-managers/) like this:

```
import unittest

from square import Square


class TestSquare(unittest.TestCase):
    def test_area(self):
        square = Square(10)
        area = square.area()
        self.assertEqual(area, 100)

    def test_length_with_wrong_type(self):
        with self.assertRaises(TypeError):
            square = Square('10')
```

If you run the test again, it will fail:

```
python -m unittest -v
```

Output:

```
test_area (test_square.TestSquare) ... ok
test_length_with_wrong_type (test_square.TestSquare) ... FAIL

======================================================================
FAIL: test_length_with_wrong_type (test_square.TestSquare)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\python-unit-testing\test_square.py", line 13, in test_length_with_wrong_type
    with self.assertRaises(TypeError):
AssertionError: TypeError not raised

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

The `test_length_with_wrong_type()` method expected that the `Square` constructor raises a `TypeError` exception. However, it didn't.

To pass the test, you need to raise an exception if the type of the `length` property is not `int` or `float` in the `Square` constructor:

```
class Square:
    def __init__(self, length) -> None:
        if type(length) not in [int, float]:
            raise TypeError('Length must be an integer or float')
```

```
        self.length = length

    def area(self):
        return self.length * self.length
```

Now, all the tests pass:

```
python -m unittest -v
```

Output:

```
test_area (test_square.TestSquare) ... ok
test_length_with_wrong_type (test_square.TestSquare) ... ok

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

OK
```

The following example adds a test that expects a `ValueError` exception if the `length` is zero or negative:

```
import unittest
```

```
from square import Square



class TestSquare(unittest.TestCase):
    def test_area(self):
        square = Square(10)
        area = square.area()
        self.assertEqual(area, 100)

    def test_length_with_wrong_type(self):
        with self.assertRaises(TypeError):
            square = Square('10')

    def test_length_with_zero_or_negative(self):
        with self.assertRaises(ValueError):
            square = Square(0)
            square = Square(-1)
```

If you run the test, it'll fail:

```
python -m unittest -v
```

Output:

```
test_area (test_square.TestSquare) ... ok
test_length_with_wrong_type (test_square.TestSquare) ... ok
test_length_with_zero_or_negative (test_square.TestSquare) ... FAIL

======================================================================
FAIL: test_length_with_zero_or_negative (test_square.TestSquare)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "D:\python-unit-testing\test_square.py", line 17, in test_length_with_zero_or_negative
    with self.assertRaises(ValueError):
AssertionError: ValueError not raised

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
```

To make the test pass, you add another check to the `Square()` constructor:

```
class Square:
    def __init__(self, length) -> None:
        if type(length) not in [int, float]:
            raise TypeError('Length must be an integer or float')
        if length < 0:
```

```
            raise ValueError('Length must not be negative')

        self.length = length

    def area(self):
        return self.length * self.length
```

Now, all three tests pass:

```
python -m unittest -v
```

Output:

```
test_area (test_square.TestSquare) ... ok
test_length_with_wrong_type (test_square.TestSquare) ... ok
test_length_with_zero_or_negative (test_square.TestSquare) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.001s

OK
```

## Summary

- A unit test is an automated test that verifies a small piece of code, executes fast, and executes in an isolated manner.
- Use the `unittest` module to perform unit testing.
- Create a class that inherits from the `unittest.TestCase` class to make a test case.
- Use the `assertEqual()` method to test if two values are equal.
- Use the `assertRaises()` method in a context manager to test expected exceptions.
- Use the `python -m unittest -v` command to run a test.