# Theory: Unittest in more detail



22 minutes 0 / 5 problems solved



Testing is a very important part of writing any software application or product. Some developers write tests for the application before the code. This is called **test-driven development (TDD)**. We have special Python libraries that can be useful and the `unittest` is one of them.

We have already learned the basics of the `unittest` testing framework — we've learned how to create test cases and tests, how to use assert methods, how to run tests, and how to read the message about the result of the tests. However, the `unittest` framework provides a lot of other tools to make testing easier. We will learn some advanced features of the library in this topic.

## §1. setUp and tearDown methods

Sometimes we need to create class instances, temporary files, or access the web in the tests. These resources are called **fixtures**. Suppose you write several tests for the same class. You may think that you will have to create an instance of that class manually for each test. It can be a repetitive and time-consuming process. The `unittest.TestCase` class (namely, the `setUp()` and `tearDown()` methods) provides an easy mechanism to configure and clean up any fixtures.

The code inside the `setUp()` method is executed ***before*** every test method and the code inside the `tearDown()` method is executed ***after*** every test method. These methods save us from having to write the same code for each test. Now let's see these methods in action!

First, we will create a class with some methods that perform some mathematical operations. This is what `calculator.py` may contain:

```
# this code is in the calculator.py file

class Calculator:

    def __init__(self, first, second):
        self.first = first
        self.second = second

    def add(self):

        """ Addition """

        return self.first + self.second


    def subtract(self):

        """ Subtraction """

        return self.first - self.second
```

The `Calculator` class has two arguments, the `first` and the `second`, and has two methods, `add()` and `subtract()`. Now let's write tests for the `Calculator` class using the `setUp()` and `tearDown()` methods.

To get the clear picture about the order of the methods, we will add a print statement with the name of the method at the end. In the `setUp()` method we create an instance of the `Calculator` class with two arguments, 5 and 1. We add the `tearDown()` method, but we don't write anything there, because we don't need it to perform any actions for now. After that, we will write the tests for the `add()` and `subtract()` methods.

Current topic:

Unittest in more detail

Topic depends on:

✕ Unit testing in Python

Table of contents:

```
# this code is in the test_calculator.py file

import unittest
import calculator


class TestCalculator(unittest.TestCase):

    def setUp(self):

        self.calc = calculator.Calculator(5, 1)

        print('setUp method')



    def tearDown(self):

        print('tearDown method')



    def test_add(self):

        self.assertEqual(self.calc.add(), 6)

        self.calc.first = 8

        self.calc.second = 2

        self.assertEqual(self.calc.add(), 10)

        print('test_add method')



    def test_subtract(self):

        self.assertEqual(self.calc.subtract(), 4)

        self.calc.first = 8

        self.calc.second = 2

        self.assertEqual(self.calc.subtract(), 6)

        print('test_subtract method')
```

For the `test_add()` and the `test_subtract()`, we check whether they work with the arguments we wrote in the `setUp()` method first, if everything is ok, we change the arguments and check again.

Let's run our tests. If everything works as expected, we would get the following message:

```
setUp method
test_add method
tearDown method
.setUp method
test_subtract method
tearDown method
.
----------------------------------------------------------------------
Ran 2 tests in 0.003s


OK
```

The print statements show the order of methods' execution. The `setUp()` method is executed before each test method, and the `tearDown()` is executed after each test method.

In both test methods, we checked the expected output for the arguments we wrote in `setUp()` first. This is because the `setUp()` method runs before each test method. So even if we change the arguments in the `test_add()` and

check that they have changed, in `test_subtract()` we write the assertion for the initial arguments.

The `tearDown()` method can be used when we work with files. For example, in the `setUp()` we can create a file that we are going to use in a test method, and then we can automatically delete it using `tearDown()` after the test method was executed.

The `unittest` also provides possibilities for class and module fixtures.

## §2. Command-line options

You know that tests can be executed from the command line, but we have some cool features for you.

Sometimes the module with the tested unit and the file with tests are located in different directories. For instance, you have `test_calculator.py` that is located in a different directory as the `calculator.py`. To do this, we need to run the command line from the directory of `calculator.py` and specify the path to `test_calculator.py`:

```
python -m unittest C:\Users\User\Hyperskill\test_calculator.py
```

When the tests are located in the same directory as the tested unit, we can specify not only the file we want to run but a test case and even a test method. Let's see how it works with our `test_calculator.py`.

- If we want to specify a test case (our code has only one case, but there are more usually), we should write the name of the test case in the command line:

  ```
  python -m unittest test_calculator.TestCalculator
  ```

  In this case, only the tests from the `test_calculator` module in the `TestCalculator` test case will be executed.

- If you want to execute only one test method, for example, the `test_add()`, add the name of this method:

  ```
  python -m unittest test_calculator.TestCalculator.test_add
  ```

  It comes in handy when you are dealing with one particular test method that you need to check. Note that the `setUp()` and `tearDown()` methods will also be executed.

- We can also run several test modules at once — you just need to type their names in the command line one after another:

  ```
  python -
m unittest test_calculator.py test_calculator2.py test_calculator3.py
  ```

  In this case, all tests from these three modules will be executed in one run!

- It is possible to combine all these options. For example, we can run all tests from `test_caculator.py` file and only `test_add()` from `test_calculator2.py` like this:

  ```
  python -
m unittest test_calculator.py test_calculator2.TestCalculator.test_add
  ```

There are several command-line options that allow us to modify the testing process and the output we receive.

- We can run tests with more detail, or higher **verbosity**. To get the details for each test, we need to type '-v' (verbosity) before the name of the test module:

  ```
  python -m unittest -v test_calculator.py
  ```

  We will get the following output:

```
test_add (test_calculator.TestCalculator) ... ok
test_subtract (test_calculator.TestCalculator) ... ok

----------------------------------------------------------------------
--
Ran 2 tests in 0.003s

OK
```

- Another useful feature accessible from the command line is when the tests are stopped after a failure. Just type '-f' before the name of the module:

```
python -m unittest -f test_calculator.py
```

The list of the command-line options is bigger, we only described the most useful and commonly-used features. If you feel that you need to know more about this, you can type '-h' (help) and read the explanations:

```
python -m unittest -h
```

# §3. Summary

We've learned more about the `unittest` framework.

- The code inside the `setUp()` method is executed before each test method.
- The code inside the `tearDown()` method is executed after each test method.
- We can specify not only a module, but also a test case and a test method if we use the command line.
- We can get details for each test method by typing '-v', or stop the tests after the first failure or error by typing '-f' in the command line.

Report a typo

**14** users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (1) Hints (0) Useful links (0) Show discussion