Python → Testing and debugging → Unit testing in Python

Theory: Unit testing in Python

31 minutes 0 / 5 problems solved

Skip this topic Start practicing

461 users solved this topic. Latest completion was about 2 hours ago.

In this topic, we are going to learn about unit testing in Python. First, let's go back and recap what unit testing is. A **unit** is a small part of the code that performs one task, and we write tests to determine whether the unit works correctly.

In general, units take input data and generate output data. So with unit testing, we know the input and the expected output, and we just compare the actual output with the expected. We can write numerous tests checking most of the case scenarios. Unit testing enables developers to detect bugs at early stages and notice if the code works incorrectly after changes.

We can do unit testing either manually or automatically. It is rarely done manually because it is a very time-consuming task. Python provides a lot of instruments for automated unit testing. `unittest` is the most popular test framework in Python, so we are going to learn how to use it in this topic. But it is not the only tool in Python for unit testing; you can also use, for example, nose, pytest, or doctest.

§1. Getting started

`unittest` is a module from the standard library with a great set of tools for writing tests. To see how it works, we will write a simple calculator and then test this program. This is the code of our calculator:

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

def add(a, b):
    """ Addition """
    return a + b


def multiply(a, b):
    """ Multiplication """

    return a * b




def subtract(a, b):

    """ Subtraction """

    return a - b




def divide(x, y):

    """ Division """

    if y == 0:

        raise ValueError('Can not divide by zero!')

    return x / y
```

Now, the `calculator.py` module contains four different functions that perform basic arithmetic operations: addition, multiplication, subtraction, and division. We are going to write unit tests to check that these functions work as expected.

Current topic:

Unit testing in Python

Topic depends on:

✗ Unit testing

✓ Load module Stage 1 5★

✓ Methods Stage 1 3★

✓ Exception handling 3★

Topic is required for:

Unittest in more detail

Table of contents:

It is better to store tests in a separate file, and it is advised to start the name of the file with the *test*. So we create a new file `test_calculator.py` and import the `unittest` module and the module we are going to test, that is the `calculator`. Note that the tested module should be in the same directory.

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

import unittest
import calculator
```

§2. Time to test

Now, we are ready to test our program. To do this, we will write one or several test cases. A **test case** is a basic unit of testing, it checks that the tested unit produces the right output when given various kinds of input. We create a test case by subclassing the general `unittest.TestCase` class:

```
class TestCalculator(unittest.TestCase):  # a test case for the calculator.py module
```

In our case, the tested unit is the whole `calculator.py` module, but we could write a separate test case for each function. In Python, a tested unit can be a class, a method, or a function.

All tests will now be defined as methods inside this class. Let's write the simplest test to check the result of our `add()` function:

```
class TestCalculator(unittest.TestCase):  # a test case for the calculator.py module

    def test_add(self):
        # tests for the add() function
        self.assertEqual(calculator.add(6, 4), 10)
        self.assertEqual(calculator.add(6, -4), 2)
        self.assertEqual(calculator.add(-6, 4), -2)
        self.assertEqual(calculator.add(-6, -4), -10)
```

The names of the test methods must start with the *test*. Otherwise, it is not going to work properly.

In the example above, we use the `assertEqual()` method from the `unittest.TestCase` class: it checks that the two given arguments are equal, and if otherwise, we will get an `AssertionError` and the test will be marked as failed.

Note that inside one test we check several cases, how the function works when two positive numbers are given, one positive and one negative, and two negative numbers. It is important that we check all possible border cases and all cases when something can go wrong.

The tests for the `multiply()` and `subtract()` functions will look similar.

PyCharm allows you to create tests in a simpler manner, you just need to right-click the name of the tested function or class and choose the option **Go To**, and then **Test**. You can learn more about writing tests with PyCharm in this tutorial.

§3. Assert methods

The `unittest.TestCase` class provides special assert methods that are used for testing. You have seen one of them in the example above, we checked that the result of the addition is correct with the help of the `assertEqual()` method.

All assert methods accept a message argument that, when specified, is used as the error message if the test fails:

```
class TestCalculator(unittest.TestCase):  # a test case for the calculator.py module

    def test_add(self):
        # tests for the add() function
        self.assertEqual(calculator.add(6, 4), 10, 'Error when adding two positive numbers')
```

You will see how error messages are displayed in the following sections.

Now, let's write tests for the `divide()` function. We can write most of the checks using the already known `assertEqual()` method, so we are not going to mention them. However, our function is also supposed to raise an exception when the divider is 0. We must check it as well and will do it with the help of the `assertRaises()` method. It works a bit differently from the `assertEqual()`, and two ways to use this method are below.

1. We can pass to the function several arguments — the exception that we expect (`ValueError`), the function that we test (`divide`), and then all arguments that the function takes (`5`, `0`):

```
class TestCalculator(unittest.TestCase):  # a test case for the calculator.py module

    def test_divide(self):
        # tests for the divide() function
        # ...
        self.assertRaises(ValueError, calculator.divide, 5, 0)
```

2. Alternatively, we can use a context manager, within which we call the tested function as we have done it before:

```
class TestCalculator(unittest.TestCase):  # a test case for the calculator.py module

     def test_divide(self):
        # tests for the divide() function
        # ...
        with self.assertRaises(ValueError):
            calculator.divide(5, 0)
```

All other assert methods are similar to the `assertEqual()` method, so we are not going to discuss them separately. In the table below, we list all widely-used methods:

Method	What it checks
`assertEqual(a, b)`	a == b
`assertNotEqual(a, b)`	a != b
`assertTrue(x)`	bool(x) is True
`assertFalse(x)`	bool(x) is False
`assertIsNone(x)`	x is None
`assertIsNotNone(x)`	x is not None
`assertGreater(a, b)`	a > b
`assertLess(a, b)`	a < b
`assertIsInstance(a, b)`	isinstance(a, b)
`assertRaises(exception, function, arguments)`	The function raises the exception when given the arguments

If you want more information about the assert methods, you can read the official Python documentation.

§4. Running tests

Once the tests are ready, we should run them and check the code. However, if you run `test_calculator.py` as a usual Python file, you are not going to get any information about the result of testing. To see the results, you should run the file from the command line, from the directory where the `test_calculator.py` is located. You need to enter either of the commands:

```
python -m unittest
```

```
python -m unittest test_calculator
```

If we do not specify the name of the test file, only files which start with the "*test*" will be executed.

There is also an easier way to run the tests right from the editor and get the message. We just need to add at the end of our code the following lines:

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

Now, if we run the module directly (not imported in some other module), then all our tests will be collected and executed. In commands, `'-m'` does exactly the same.

After that, you will get a message with information about the tests. We talk about these messages in detail in the next section.

§5. Test outcomes

When the tests are executed, we get a message which provides us with information about the result of testing. For example, if we run the tests we have written for our calculator, we will see the following message:

```
....
----------------------------------------------------------------------
Ran 4 tests in 0.001s

OK
```

`OK` means that all tests went well, and so do the dots that correspond to the succeeded test cases. So, from this message, we know that:

- 4 tests were executed;
- all tests succeeded.

Now let's imagine that we made a typo in the `add()` function, and accidentally put '-' instead of '+':

```
def add(a, b):
    return a - b
```

Then, we'll get the following message:

```
F...
======================================================================
FAIL: test_add (__main__.TestCalculator)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\...\test_calculator.py", line 11, in test_add
    self.assertEqual(calculator.add(6, 4), 10, 'Error when adding two positive numbers')
AssertionError: 2 != 10 : Error when adding two positive numbers

----------------------------------------------------------------------
Ran 4 tests in 0.003s

FAILED (failures=1)
```

This message tells us:

- 4 tests were executed;
- 3 tests passed (dots);
- one test failed (the letter `'F'` and the number of failed test explicitly shown in the last line);
- where something went wrong (the 11th line; in the `test_add` method);
- what went wrong (assertion failure).

Note that together with the `AssertionError`, we see the message that we specified in the code: "*Error when adding two positive numbers*". It helps us understand what the error was.

The tests are executed alphabetically, so the order of dots and letters in the first line does not correspond to the order of tests in our code.

There is also the third possible outcome — ERROR. The errors occur when a test raises an exception *other than* `AssertionError`. In such cases, we see the letter `'E'` in the first line and the information about the occurred problem.

Let's say we wrote in the tests for the `divide()` function the following assertion:

```
def test_divide(self):
    # tests for the divide function
    # ...
    self.assertEqual(calculator.divide(10, 0), 0)
```

Then we would get the following outcome:

```
.E..
======================================================================
FAIL: test_divide (__main__.TestCalculator)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\...\test_calculator.py", line 28, in test_divide
    self.assertEqual(calculator.divide(10, 0), 0)
  File "C:\Users\...\calculator.py", line 16, in divide
    raise ValueError('Can not divide by zero!')

ValueError: Can not divide by zero!
```

```
----------------------------------------------------------------------
Ran 4 tests in 0.003s

FAILED (errors=1)
```

First, it tells us that an error occurred in the line No. 28, in the `test_divide` (the letter `'E'` is the second of all the dots). Further in the message, we can see that the `divide` method raises a `ValueError` if we try to divide it by zero. As a result, the `ValueError` does take place, which is not `AssertionError`, so the test is considered neither failed nor passed.

§6. Summary

In this topic, we have discussed the basics of unit testing in Python using the `unittest` framework. The main points to remember are as follows:

- We write tests in a separate file, in the very beginning of which we import `unittest` and the tested module.
- To create a test case, we subclass the `unittest.TestCase` class.
- We use *assert methods* for writing tests. The most commonly used assert method is `assertEqual()`.
- Tests can result in 3 possible ways:
  - Success — the test passes, everything worked as expected;
  - Failure — the test doesn't pass and raises an AssertionError exception, meaning that the assertion failed;
  - Error — the test raises an exception other than AssertionError.

Report a typo

56 users liked this theory. **2** didn't like it. What about you?

Start practicing

Comments (2) Hints (0) Useful links (0) Show discussion