
ICTAcademy
INFORMATION AND COMMUNICATIONS TECHNOLOGY
Testing in Python

Testing in python - Intro

Unit tests are segments of code written to test other pieces of code, typically a single function or method, that we refer to as a unit.

They are a very important part of the software development process, as they help to ensure that code works as intended and catch bugs early on.

Also, testing is a best practice that can save time and money by finding and fixing issues before they cause major problems.

Testing in python - Intro

Python has two main frameworks to make unit testing easier: `unittest` and `PyTest`.

The first one has been part of Python's standard library since Python 2.1 and that's the one we're focused on in this lecture.

The assert Statement

The assert statement is a built-in statement in Python used to, as the name says, assert if a given condition is true or not.

If the condition is true, nothing happens, but if it's not true, an error is raised. Although, at first, it may look like the try and except clauses, they are completely different, and assert should not be used for error handling but for debugging and testing reasons.

As an example, the condition in the line below is true and, therefore, it does not output or return anything:

assert 1 > 0

However, if we change this condition so it becomes false, we get an AssertionError:

assert 1 < 0

```
---------------------------------------------------------------------------
AssertionError
Traceback (most recent call last)
<ipython-input-2-2d19dbe67b58> in <module>
----> 1 assert 1 < 0
AssertionError:
```

The assert Statement

Notice that in the last row of the error message there isn't an actual message after AssertionError:.
That's because the user should pass this message.
Here's how:

```
n = 0
assert 1 < n, 'The Condition is False'
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-3-e335e3eb84ff> in <module>
      1 n = 0
----> 2 assert 1 < n, 'The Condition is False'
AssertionError: The Condition is False
```

The assert Statement - Syntax

So, the basic syntax for using assert is the following:

assert <condition being tested>, <error message to be displayed>

Assert is very simple to use.

Understanding it is critical for testing purposes, as we'll see in the following sections.

Unittest Module

The `unittest` module is a framework designed to make our lives easier when it comes to testing code. The module works based on some important object-oriented concepts, and that's why you need to understand the basics of classes and methods in Python.

Unittest Module - TestCase

A test case is considered a single unit of testing, and it's represented by the `TestCase` class.

Among the numerous tools provided by `unittest` that allow us to test code, this class is one of the most important ones. It's used as a base class to create our own test cases that enable us to run multiple tests at once.

Unittest Module - Asserts Statements

Although we've seen the importance of the Python `assert` statement in the last section, **it won't be used here.**

The `TestCase` class provides several of its own assert methods that work just like the `assert` statement but for specific types of assertions.

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

Implementing Unittests - First Step

Let’s write a calculator class in python

```
class Calculations:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def get_sum(self):
        return self.a + self.b

    def get_difference(self):
        return self.a - self.b

    def get_product(self):
        return self.a * self.b

    def get_quotient(self):
        return self.a / self.b
```

Implementing Unittests - Second Step

Now let’s write a series of test for the class!

```
import unittest
from code.my_calculations import Calculations

class TestCalculations(unittest.TestCase):

    def test_sum(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_sum(), 10, 'The sum is wrong.')

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

Implementing Unittests - Second Step

Now let’s write a series of test for the class!

```
import unittest
from code.my_calculations import Calculations

class TestCalculations(unittest.TestCase):

    def test_sum(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_sum(), 10, 'The sum is wrong.')

    def test_diff(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_difference(), 6, 'The difference is wrong.')

    def test_product(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_product(), 16, 'The product is wrong.')

    def test_quotient(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_quotient(), 4, 'The quotient is wrong.')

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

Implementing Unittests - Second Step

Be Aware, it isn't an accident that all the methods' names start with the word `test`.

This is a convention we use so that `unittest` can identify the tests it's supposed to run.

```
import unittest
from code.my_calculations import Calculations

class TestCalculations(unittest.TestCase):

    def test_sum(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_sum(), 10, 'The sum is wrong.')

    def test_diff(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_difference(), 6, 'The difference is wrong.')

    def test_product(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_product(), 16, 'The product is wrong.')

    def test_quotient(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_quotient(), 4, 'The quotient is wrong.')

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

Implementing Unittests - Second Step

Be Aware, it isn't an accident that all the methods' names start with the word `test`.

This is a convention we use so that `unittest` can identify the tests it's supposed to run.

For instance, the following code runs only three tests:

```
import unittest
from code.my_calculations import Calculations

class TestCalculations(unittest.TestCase):

    def not_a_test_sum(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_sum(), 10, 'The sum is wrong.')

    def test_diff(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_difference(), 6, 'The difference is wrong.')

    def test_product(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_product(), 16, 'The product is wrong.')

    def test_quotient(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_quotient(), 4, 'The quotient is wrong.')

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

Optimizing Unittest - setUp Method

Now that we understand the basics of unit testing with the `unittest` module, let's optimize our code a bit. You probably have noticed that inside each test we initialized an object of the `Calculations` class, which will be tested.

We can avoid that by creating a `setUp` method!

```python
import unittest
from code.my_calculations import Calculations

class TestCalculations(unittest.TestCase):

    def test_sum(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_sum(), 10, 'The sum is wrong.')

    def test_diff(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_difference(), 6, 'The difference is wrong.')

    def test_product(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_product(), 16, 'The product is wrong.')

    def test_quotient(self):
        calculation = Calculations(8, 2)
        self.assertEqual(calculation.get_quotient(), 4, 'The quotient is wrong.')

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

Optimizing Unittest - setUp Method

The `TestCase` class already has a `setUp` method that runs before each test.

So what we'll do when creating a new one is actually overwrite the default method with our own.

This is the code with this new method implemented:

```
import unittest
from code.my_calculations import Calculations

class TestCalculations(unittest.TestCase):

    def setUp(self):
        self.calculation = Calculations(8, 2)

    def test_sum(self):
        self.assertEqual(self.calculation.get_sum(), 10, 'The sum is wrong.')

    def test_diff(self):
        self.assertEqual(self.calculation.get_difference(), 6, 'The difference is wrong.')

    def test_product(self):
        self.assertEqual(self.calculation.get_product(), 16, 'The product is wrong.')

    def test_quotient(self):
        self.assertEqual(self.calculation.get_quotient(), 4, 'The quotient is wrong.')

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

Unittest from command line

To run all the tests in a directory we must use the command `python -m unittest` to launch the discovery mode in `unittest` that will look for the tests inside the current directory.

However, for the tests to run, we have to follow some naming conventions:

- The name of each file containing tests has to start with `test`
- All the tests have to be methods of class based on the `TestCase` class.
- The names of all these methods have to start with the word `test`
- The directory must be an importable module, which means it should contain an `__init__.py` file.

Unittest from command line - Running multiple files

Using the command `python -m unittest` and adding the flag -v allow us to display cool infos about the tests!

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

```
test_diff (teste.test.TestCalculations) ... ok
test_product (teste.test.TestCalculations) ... ok
test_quotient (teste.test.TestCalculations) ... ok
test_sum (teste.test.TestCalculations) ... ok
test_isupper (teste.test_str.TestStringMethods) ... ok
test_split (teste.test_str.TestStringMethods) ... ok
test_upper (teste.test_str.TestStringMethods) ... ok
----------------------------------------------------------------------
Ran 7 tests in 0.002s
OK
```

Unittest from command line - Running multiple files

We can also specify a single test file inside a folder and run just that using the command

```
python -m unittest -v tests.test
```

Or we can do the same with a single test function inside a test file

```
python -m unittest -v tests.test.TestCalculations.test_diff
```

The results in this case would be

```
test_diff (teste.test.TestCalculations) ... ok

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

OK
```