Unit Testing in Python — The Basics

Increase the quality, trustworthiness and flexibility of your code base

Martin Thoma
Jun 29, 2020 · 6 min read ★

Image Source: Mohamed Hassan

Unit testing is the number one skill which separates people who just finished their degrees from people with practical experience. Especially for Python, that's a shame as it is trivial to learn this skill.

In this article, you will learn how to write and run unit tests in Python as well as some interesting pytest plugins I usually use. Let's get started.

The most basic Unit Test

A unit test is atomic- it just tests one unit of code. Typically one function or one method of a class. As an example, let's say we want to test `math_functions.py` which contains the Fibonacci function and a function for the Collatz sequence:

We want to test this function. I will explain the reasons for testing and what testing means later. For now, let's just say we want to avoid programming errors.

First, create a file `test_math_functions.py` :

Now, you have to install pytest:

```
$ pip install pytest
```

And run it:

```
$ pytest
============ test session starts ====================================
platform linux -- Python 3.8.1, pytest-5.4.3, py-1.9.0, pluggy-
0.13.1
rootdir: /home/moose/GitHub/MartinThoma/algorithms/medium/unit-
testing
collected 5 items

test_math_functions.py .....                                    [100%]

============ 5 passed in 0.03s ======================================
```

Awesome! You can see that it took 0.03 seconds to execute. There are 5 dots after the `test_math_functions.py` . Those indicate that 5 tests were executed and successful.

Let's break one test, e.g. `test_fib_3` by setting `assert fib(3) == 1337` . Then you see this:

```
============ test session starts ================================
platform linux -- Python 3.8.1, pytest-5.4.3, py-1.9.0, pluggy-
0.13.1
rootdir: /home/moose/GitHub/MartinThoma/algorithms/medium/unit-
```

```
testing
collected 5 items

test_math_functions.py ..F..
[100%]

================= FAILURES ===================================
_________________ test_fib_3 ________________________________

def test_fib_3():
>       assert fib(3) == 3
E       assert 2 == 3
E        +  where 2 = fib(3)

test_math_functions.py:14: AssertionError
============ short test summary info ============================
FAILED test_math_functions.py::test_fib_3 - assert 2 == 3
============ 1 failed, 4 passed in 0.03s ========================
```

Great. Now you know how to write a unit test.

Vocabulary

The *units* we are testing in the section above are functions — `fib` and `next_collatz_element` .

We have 5 *unit tests*; all of them in `test_math_functions.py` : The `test_*` functions.

The `pytest` command-line executable is called a *test runner*. It executes (runs) the tests.

A *test suite* is an arbitrary collection of tests. Usually, you mean all tests.

Why do we test at all?

- **Trust**: You checked at least some cases if they work. So others can have more trust in the quality of your work and you can also put more trust in it.
- **Breaking Changes**: For a bigger project, it is sometimes hard to have every part in mind. By writing tests, you make it easier to change something and see if / where things break. This does not only help you but also team members. Including once that are not there yet.
- **Code Style**: When you know that you have to write tests, you write some things slightly differently. Those slight differences usually improve the coding style.

Sometimes, they are crucial. For example, if you have to thoroughly test your code you will make smaller chunks.

- **Documentation**: Some test cases show a little bit of how the code is intended to be used.

Test Coverage

I hope at this point we agree that having tests is a good idea. But how many tests do you need? When did you test everything?

A group of measures for this is the *test coverage*. There are two relevant types of test coverage: Line coverage and branch coverage.

If you look at the Collatz function from above, there are 4 lines to test:

If I execute `next_collatz_element(4)` , then it will execute lines 1–3. Line 4 and 5 will not be hit. This means a unit test like that could not detect an issue on line 4 or 5. It only covers 3 of 5 lines. One says that it has 60% line coverage.

But sometimes 100% line coverage is not enough. Take a look at this example:

If you test `greet("Angela", "Merkel")` you will have 100% line coverage. But you miss that if the last name is not given, the return value is `None` . In the given test, the if-statement in line 2 always evaluates to “True”. You don’t cover a branch in the execution graph. So you have only 50% branch coverage.

`pytest-cov` is a pytest plugin to measure branch coverage.

- Install it with `pip install pytest-cover`
- Use it by adding `--cov=path/to/file` or `--cov=packagename` to the pytest execution
- Get output to terminal by adding to pytest `--cov-report term`
- Get HTML output by adding `--cov-report html:tests/reports/coverage`

There are more reporting capabilities.

Good Tests

It's pretty hard to write good tests and when you measure your test coverage it is tempting to quickly write a couple of bad tests.

Worst is no testing at all.

A little bit better is a test that just executes a function but does not check if the return value/the side effects are what you expect. So you simply run it to check if the code crashes.

Happy-Tests where you check the output of the tested function and a typical input is even better. I call them *happy* because they test what you expect to get.

In contrast, an *unhappy* execution path is dealing with unwanted inputs. This is also called negative testing. You check if you actually throw an error. Not throwing an error and silently failing is bad as it hides bugs.

Property testing is pretty cool. There you don't test for single values, but you check if a property is still held. For example, the output of a factorization function can be multiplied and should equal the input.

Type Checking

If you use type annotations (which you totally should!), then you can install `pytest-mpy` . You can then automatically run mypy over your code by adding `--mypy` to your pytest command.

Linting

Code linting is the act of finding bugs, stylistic errors, and suspicious constructs from static code analysis.

There are two linters I can recommend: black and flake8. You can run them with pytest by installing `pytest-black` and `pytest-flake8` . Again, if you want to execute it just add the flag `--black` or `--flake8` to pytest:

```
$ pytest --flake8 --black
```

There is also `pytest-mccabe` which tries to find sections in the code which are too complex. This makes it easier for coworkers / your future self to understand the code.

An alternative to pytest-mccabe outside of pytest is `radon` . However, a lot of people don't like this type of test.

Doctests

Doctests are a weird but pretty awesome part of Python. Python has Docstrings — the first string within a function/class which comes directly after the signature and which is not assigned. This is not just a comment, it has meaning and can be read through the execution:

If you execute this directly, the lines 19–21 will run the doctest. The doctest looks for `>>>` within the docstrings and executes whatever follows as if it was entered in the interactive console. The next line is then the output which is compared to the output of the program.

This is pretty awesome because it makes documentation testable!

And, of course, you can also execute doctests with pytest:

```
$ pytest --doctest-modules
```

Test Execution Speed

It's important to keep the execution time of the tests low so that it doesn't feel bad to execute the test suite. I like to print the time of the 3 slowest tests which were performed. To profile the tests continuously, I simply add the durations flag:

```
$ pytest --durations=3
```

Alternatives: unittest and nose

unittest is a core Python module and as such, I would prefer to use it. unittest feels pretty similar to JUnit which I would say is a disadvantage. Python is a different language with different patterns and expectations. One weirdness is that you have to put your tests in a class, even if you don't need to setUp() or tearDown() anything. It uses camelCase for the method names which is against the Python conventions. You cannot simply `assert Expression` , but instead, have to use `self.assertEqual` ,

`self.assertTrue` , … (see the complete list of assert methods). And the error messages are not as expressive as the ones you get from Pytest.

TL;DR: unittest and nose are no alternatives. pytest is the way to go.

What's next?

In this series, we already had:

- Part 1: The basics of Unit Testing in Python
- Part 2: Patching, Mocks and Dependency Injection
- Part 3: How to test Flask applications with Databases, Templates and Protected Pages
- Part 4: tox and nox
- Part 5: Structuring Unit Tests
- Part 6: CI-Pipelines
- Part 7: Property-based Testing
- Part 8: Mutation Testing
- Part 9: Static Code Analysis — Linters, Type Checking, and Code Complexity
- Part 10: Pytest Plugins to Love

Let me know if you're interested in other topics around testing with Python.

Python Unit Testing Software Development Software Engineering Programming