

How to use code coverage in Python with pytest?

📅 April 11, 2021 (<https://breadcrumbscollector.tech/how-to-use-code-coverage-in-python-with-pytest/>)

👤 Sebastian (<https://breadcrumbscollector.tech/author/hansel/>) 📁 python

(<https://breadcrumbscollector.tech/category/python/>), testing software

(<https://breadcrumbscollector.tech/category/testing-software/>)

Basics

What is code coverage?

In the simplest words, code coverage is a measure of exhaustiveness of a test suite. 100% code coverage means that a system is fully tested.

Why bother about code coverage in Python?

Theoretically, the higher code coverage is, the fewer defects a system has. Of course, tests are not enough to catch all kinds of errors, but in this uneven battle, we need all the help we can get.

From a very mechanical perspective, the codebase is composed of individual lines. Hence, a simple formula for code coverage would be

$$\frac{(\text{number_of_code_lines_run_at_least_once_under_tests} / \text{total_number_of_lines}) * 100\%}{100\%}$$
. It is only at first sight that this formula looks reasonable. It's way too insufficient. For the purpose of this article, consider the following piece of code:

```
from dataclasses import dataclass

@dataclass
class Patient:
    age: int
    is_pregnant: bool = False
    is_regular_blood_donor: bool = False

def determine_queue_position(patient, queue):
    # initially, we assume that a patient will just join queue
    position = len(queue)

    # there are certain groups of patients that are served without
    # having to wait in a queue
    if patient.is_pregnant or patient.is_regular_blood_donor:
        position = 0

    return position
```

Why focusing on just covering lines is not enough

Now, let's assume we have a test for that:

```
def test_pregnancy_means_accessing_doctor_without_having_to_wait():
    queue = [Patient(age=25), Patient(age=44)]
    patient = Patient(age=28, is_pregnant=True)

    queue_position = determine_queue_position(patient, queue)

    assert queue_position == 0
```

This test exercises EVERY line of `determine_queue_position` function. According to our initial definition, we were able to get 100% code coverage with a single test. Yet this minimal test suite can be hardly called exhaustive! For example, we haven't tested against such patients:

- a regular blood donor
- both pregnant and regular blood donor
- neither pregnant nor regular blood donor

etc. Not to mention cases like a queue with one or more patients pregnant or being regular blood donor (the latter is not covered by implementation, so we won't be focusing on it anyway).

Types of code coverage

While the original definition of code coverage is still valid (a measure of exhaustiveness of a test suite), turns out there is a tricky part. Namely, how to assess if a test suite is actually exhaustive?

Statement coverage

We already know that a naive approach with measuring executed lines of code won't cut it. On the bright side, it is the simplest one to understand. It is formally called line or statement coverage. This one is used by default in the most complete python code coverage lib – `coverage.py` (<https://coverage.readthedocs.io/>).

Assuming we have code in `func.py` and tests in `test_func.py` files, we can see `coverage.py` (+`pytest-cov` plugin) reports 100% code coverage:



```

pytest --cov func

===== test session starts =====
platform darwin -- Python 3.9.0, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /Users/spb/Projects/private/bloggo/coverr
plugins: cov-2.11.1
collected 1 item

test_func.py . [100%]

----- coverage: platform darwin, python 3.9.0-final-0 -----
Name      Stmts   Miss  Cover
-----
func.py    11      0   100%
-----
TOTAL      11      0   100%

===== 1 passed in 0.04s =====

```

If statement coverage is so superficial, what are better alternatives?

Branch coverage

While code indeed is composed out of lines, our execution is rarely sequential from the top to the bottom. This is because of if-statements (and similar mechanisms) that steer how the execution *flows*. When there is decision-making whether to do one or another thing, we call it branching. Respectively, possible code paths are called branches.

This leads us to another type of code coverage – branch coverage. It is defined as $(\text{number_of_branches_executed_at_least_once_under_tests} / \text{all_branches}) * 100\%$. This gives us a better idea about uncovered scenarios:

```

pytest --cov func --cov-branch --cov-report term-missing

===== test session starts =====
platform darwin -- Python 3.9.0, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /Users/spb/Projects/private/bloggo/coverr
plugins: cov-2.11.1
collected 1 item

test_func.py . [100%]

----- coverage: platform darwin, python 3.9.0-final-0 -----
Name      Stmts   Miss Branch  BrPart  Cover   Missing
-----
func.py    11      0      4       1    93%    17->20
-----
TOTAL      11      0      4       1    93%

===== 1 passed in 0.05s =====

```

Branch coverage told us that we miss an if-statement at line 17th evaluates to False and the next executed line is `return position`. Covering it is a matter of testing with a patient that's not pregnant neither a regular blood donor:

```
def test_not_pregnat_teenager_not_being_blood_donor_has_to_wait_in_queue():
    queue = [Patient(age=15), Patient(age=33)]
    patient = Patient(age=13)

    queue_position = determine_queue_position(patient, queue)

    assert queue_position == 2
```

Running test suite again shows we are now fully covered (at least in terms of branch coverage):

```
pytest --cov func --cov-branch --cov-report term-missing
===== test session starts =====
platform darwin -- Python 3.9.0, pytest-6.2.3, py-1.10.0, pluggy-0.13.1
rootdir: /Users/spb/Projects/private/bloggo/coverr
plugins: cov-2.11.1
collected 2 items

test_func.py ..                                     [100%]

----- coverage: platform darwin, python 3.9.0-final-0 -----
Name          Stmts   Miss Branch BrPart  Cover   Missing
-----
func.py         11      0      4      0   100%
-----
TOTAL          11      0      4      0   100%

===== 2 passed in 0.05s =====
```

How about other test scenarios? Python code coverage still has no clue we haven't tested a regular blood donor.

Condition coverage

While branch coverage nicely catches if we missed specific paths of execution, it's indifferent to specific conditions. You certainly remember that for example `or` is evaluated lazily – if an expression on the left side is true, then the one on the right side is not even evaluated.

```
# when is_pregnant is True, then the second part won't be executed!
if patient.is_pregnant or patient.is_regular_blood_donor:
    ...
```

Condition coverage assumes that in order to achieve 100% code coverage, the test suite needs to check situations in which every expression is True and False. It means condition coverage will require us to:

- the patient is not pregnant nor a regular blood donor
- the patient is pregnant but not a regular blood donor



- the patient is both pregnant and a regular blood donor
- the patient is not pregnant but is a regular blood donor

A formula for this type of coverage could be $(\text{number_of_executed_bool_states_of_operands} / \text{number_of_all_operands} * 2) * 100\%$.

Unfortunately, there is no maintained tool in Python that will measure it for you. There was a lib called instrumental (<https://pypi.org/project/instrumental/>) but it seems it has been abandoned for years.

On the other hand, we can resort to hypothesis (<https://hypothesis.readthedocs.io/en/latest/>) (property-based testing lib) to help us generate exhaustive use cases. This would be especially helpful for more of a black-box testing without looking into guts of tested function (white-box testing).

Other types of coverage

Statement-, Branch- and Condition coverage are not all types of code coverage. If you are hungry for more, see several white papers linked at the end of the article.

Installation & configuration

I am assuming you are using pytest.

Now, if you're new to coverage and want to get your hands dirty you can get some coverage numbers fast if you:

- `pip install pytest-cov` (it depends on coverage.py so it will be installed as well)

Regarding configuration, we would certainly want to enable branch coverage. We can do this (+ few other options) using e.g. `setup.cfg`:


```
[coverage:run]
branch = True
omit = src/db/env.py,src/db/versions/* # define paths to omit

[coverage:report]
show_missing = True
skip_covered = True
```

Good practices

When to run pytest with code coverage enabled?

During build (Continuous Integration)

Running tests with coverage should definitely happen during builds, e.g. on Jenkins, Travis or whatever tool you use. We should set some required threshold for coverage. When it's not met (code coverage less than expected) we fail the build, e.g. `pytest -cov=src/ -cov-fail-under=100 tests/`. In this example, the command will fail if our coverage is lower than 100%. 

Locally

Just like during Continuous Integration, you can instrument pytest to run coverage plugin by manually appending appropriate parameters. The other option is to configure pytest to always collect coverage when it runs by using `addopts` configuration in e.g. `setup.cfg` :

```
[tool:pytest]
addopts = --cov src/
```

Personally, I advise against the second option. Why? Because collecting code coverage in Python is a considerable performance hit. If you (or anyone in your team) is using Test-First approach, then extra latency becomes an annoyance. Usually, I run small parts of the test suite when working locally in TDD cycle and then manually run the whole test suite at the end with code coverage enabled.

How much code coverage is enough?

In theory the higher code coverage, the better. I think it makes no sense to set it at 80% or 90%. I think 100% is possible with a “BUT”.

The stance on code coverage that my colleague Łukasz (<https://lukeonpython.blog/>) taught me is that one should start with 100% requirement and then exclude lines where it is not possible to achieve code coverage. It can be done using `# pragma: no cover` comment. For example, coverage will complain about abstract base classes, which is obviously a nonsense:

```
class ApiFactory(abc.ABC):
    @abc.abstractmethod
    def foo_api(self) -> FooApi: # pragma: no cover
        pass

    @abc.abstractmethod
    def bar_api(self) -> BarApi: # pragma: no cover
        pass
```

There is also an option to set excluded lines in configuration of coverage.py but it's not ideal.

Of course, the rule of 100% test coverage must be loosened in codebases where code coverage wasn't measured before. Even then it makes sense to set the expectation high. Initially, we can also exclude parts of the code.

Is 100% code coverage an intolerable burden?

Does a pursue for 100% code coverage mean writing tests for every function/class/module?

No. This is a widely held myth. If function A uses function B, then to cover both of them testing function A can be sufficient. That will largely depend on their implementation, but in general, our code is organized hierarchically, forming levels of abstraction. Then measuring code coverage is an immense help to quickly show us which parts we missed.

Testing each and every code block individually is unreasonable. It effectively makes code immutable and tests very fragile. We should be starting from higher-level tests, adding low-level ones when necessary (and code coverage will give you a great hint when you need it!). Also, be

aware of encapsulation (<https://breadcrumbscollector.tech/encapsulation-is-your-friend-also-in-python/>) and not violating it during testing.

Summary

When one wants to truly lean on their test suite, code coverage is an indispensable thing.


Although 100% code coverage may look like an unattainable goal, in my opinion, it is the only expectation that works. It really *clicks* when combined with TDD.


Further reading

- Code coverage criteria and their effect on test suite qualities (<https://publications.lib.chalmers.se/records/fulltext/217038/217038.pdf>)
- coverage.py (<https://coverage.readthedocs.io/>)
- hypothesis (<https://hypothesis.readthedocs.io/en/latest/>)

29

Share this:

 (<https://breadcrumbscollector.tech/how-to-use-code-coverage-in-python-with-pytest/?share=twitter&nb=1>)

 (<https://breadcrumbscollector.tech/how-to-use-code-coverage-in-python-with-pytest/?share=facebook&nb=1>)

The Clean Architecture in Python. How to write testable and flexible code (/the-clean-architecture-in-python-how-to-write-testable-and-flexible-code/?relatedposts_hit=1&relatedposts_origin=888&relatedposts_position=0)
April 13, 2019
In "clean architecture"

Implementing the Clean Architecture with Python - my book is here! (/implementing-the-clean-architecture-with-python-my-book-is-here/?relatedposts_hit=1&relatedposts_origin=888&relatedposts_position=1)
March 6, 2020
In "clean architecture"

Where to put all your utils in Python projects? (/where-to-put-all-your-utils-in-python-projects/?relatedposts_hit=1&relatedposts_origin=888&relatedposts_position=2)
May 23, 2021
In "good practices"

4 thoughts to “How to use code coverage in Python with pytest?”



ŁUKASZ ŻUKOWSKI ([HTTP://LUKEONPYTHON.BLOG](http://lukeonpython.blog))

April 12, 2021 at 7:23 pm (<https://breadcrumbscollector.tech/how-to-use-code-coverage-in-python-with-pytest/#comment-5923>)

