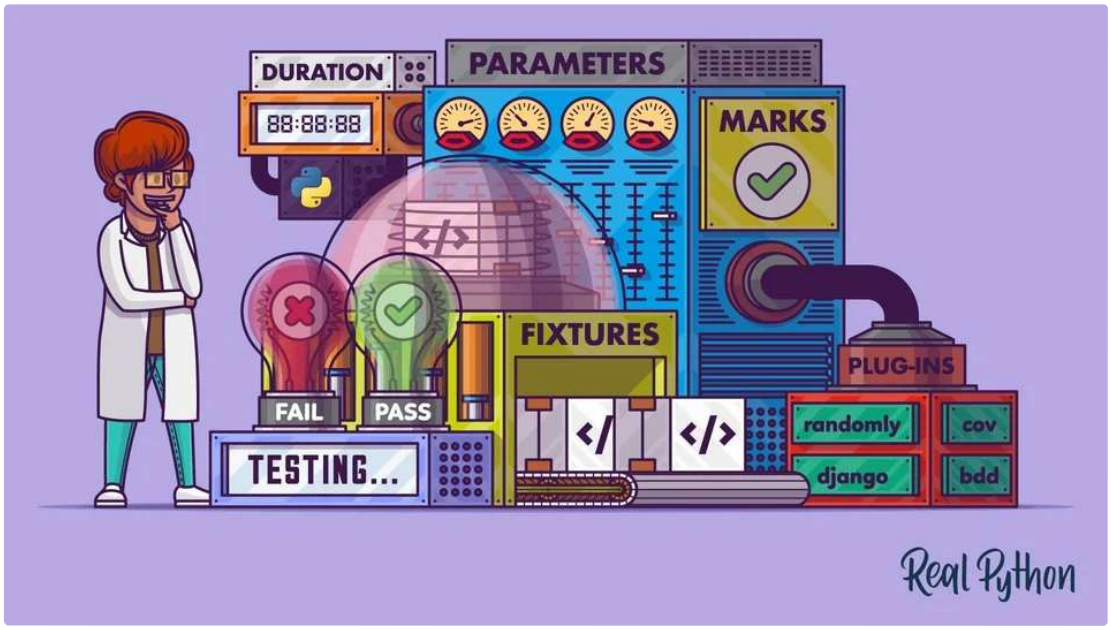

Effective Python Testing With Pytest

by Dane Hillard Apr 20, 2020 21 Comments intermediate python testing

Table of Contents

Testing your code brings a wide variety of benefits. It increases your confidence that the code behaves as you expect

Testing your code brings a wide variety of benefits. It increases your confidence that the code behaves as you expect and ensures that changes to your code won't cause regressions. Writing and maintaining tests is hard work, so you should leverage all the tools at your disposal to make it as painless as possible. `pytest` is one of the best tools you can use to boost your testing productivity.

In this tutorial, you'll learn:

- What **benefits** `pytest` offers
- How to ensure your tests are **stateless**
- How to make repetitious tests more **comprehensible**
- How to run **subsets** of tests by name or custom groups
- How to create and maintain **reusable** testing utilities

How to Install `pytest`

To follow along with some of the examples in this tutorial, you'll need to install `pytest`. As with most Python packages, you can install `pytest` in a virtual environment from PyPI using `pip`:

Shell

```
$ python -m pip install pytest
```

The `pytest` command will now be available in your installation environment.

What Makes `pytest` So Useful?

If you've written unit tests for your Python code before, then you may have used Python's built-in **`unittest`** module. `unittest` provides a solid base on which to build your test suite, but it has a few shortcomings.

A number of third-party testing frameworks attempt to address some of the issues with `unittest`, and pytest has proven to be one of the most popular. `pytest` is a feature-rich, plugin-based ecosystem for testing your Python code.

If you haven't had the pleasure of using `pytest` yet, then you're in for a treat! Its philosophy and features will make your testing experience more productive and enjoyable. With `pytest`, common tasks require less code and advanced tasks can be achieved through a variety of time-saving commands and plugins. It will even run your existing tests out of the box, including those written with `unittest`.

As with most frameworks, some development patterns that make sense when you first start using `pytest` can start causing pains as your test suite grows. This tutorial will help you understand some of the tools `pytest` provides to keep your testing efficient and effective even as it scales.

Less Boilerplate

Most functional tests follow the Arrange-Act-Assert model:

1. **Arrange**, or set up, the conditions for the test
2. **Act** by calling some function or method
3. **Assert** that some end condition is true

Testing frameworks typically hook into your test's assertions so that they can provide information when an assertion fails. `unittest`, for example, provides a number of helpful assertion utilities out of the box. However, even a small s of tests requires a fair amount of boilerplate code.

Imagine you'd like to write a test suite just to make sure `unittest` is working properly in your project. You might want to write one test that always passes and one that always fails:

Python

```
# test_with_unittest.py

from unittest import TestCase

class TryTesting(TestCase):
    def test_always_passes(self):
        self.assertTrue(True)

    def test_always_fails(self):
        self.assertTrue(False)
```

You can then run those tests from the command line using the `discover` option of `unittest`:

Shell

```
$ python -m unittest discover
F.
======================================================================
FAIL: test_always_fails (test_with_unittest.TryTesting)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/.../test_with_unittest.py", line 9, in test_always_fails
    self.assertTrue(False)
AssertionError: False is not True

----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

As expected, one test passed and one failed. You've proven that `unittest` is working, but look at what you had to do:

1. Import the `TestCase` class from `unittest`
2. Create `TryTesting`, a subclass of `TestCase`
3. Write a method in `TryTesting` for each test
4. Use one of the `self.assert*` methods from `unittest.TestCase` to make assertions

That's a significant amount of code to write, and because it's the minimum you need for *any* test, you'd end up writing the same code over and over. `pytest` simplifies this workflow by allowing you to use Python's `assert` keyword directly:

Python

```
# test_with_pytest.py

def test_always_passes():
    assert True

def test_always_fails():
    assert False
```

That's it. You don't have to deal with any imports or classes. Because you can use the `assert` keyword, you don't need to learn or remember all the different `self.assert*` methods in `unittest`, either. If you can write an expression that you expect to evaluate to `True`, then `pytest` will test it for you. You can run it using the `pytest` command:

Shell

```
$ pytest
================== test session starts ============================
platform darwin -- Python 3.7.3, pytest-5.3.0, py-1.8.0, pluggy-0.13.0
rootdir: /.../effective-python-testing-with-pytest
collected 2 items

test_with_pytest.py .F                                          [100%]
```

```
======================= FAILURES ========================================
_____________________ test_always_fails ________________________________

    def test_always_fails():
>       assert False
E       assert False

test_with_pytest.py:5: AssertionError
============== 1 failed, 1 passed in 0.07s ========================
```

`pytest` presents the test results differently than `unittest`. The report shows:

1. The system state, including which versions of Python, `pytest`, and any plugins you have installed
2. The `rootdir`, or the directory to search under for configuration and tests
3. The number of tests the runner discovered

The output then indicates the status of each test using a syntax similar to `unittest`:

- **A dot (`.`)** means that the test passed.
- **An `F`** means that the test has failed.
- **An `E`** means that the test raised an unexpected exception.

For tests that fail, the report gives a detailed breakdown of the failure. In the example above, the test failed because `assert False` always fails. Finally, the report gives an overall status report of the test suite.

Here are a few more quick assertion examples:

Python

```python
def test_uppercase():
    assert "loud noises".upper() == "LOUD NOISES"

def test_reversed():
    assert list(reversed([1, 2, 3, 4])) == [4, 3, 2, 1]

def test_some_primes():
    assert 37 in {
        num
        for num in range(1, 50)
        if num != 1 and not any([num % div == 0 for div in range(2, num)])
    }
```

The learning curve for `pytest` is shallower than it is for `unittest` because you don't need to learn new constructs for most tests. Also, the use of `assert`, which you may have used before in your implementation code, makes your tests more understandable.

State and Dependency Management

Your tests will often depend on pieces of data or test doubles for some of the objects in your code. In `unittest`, you might extract these dependencies into `setUp()` and `tearDown()` methods so each test in the class can make use of them. But in doing so, you may inadvertently make the test's dependence on a particular piece of data or object entirely **implicit**.

Over time, implicit dependencies can lead to a complex tangle of code that you have to unwind to make sense of your tests. Tests should help you make your code more understandable. If the tests themselves are difficult to understand, then you may be in trouble!

`pytest` takes a different approach. It leads you toward **explicit** dependency declarations that are still reusable thanks to the availability of fixtures. `pytest` fixtures are functions that create data or test doubles or initialize some system state for the test suite. Any test that wants to use a fixture must explicitly accept it as an argument, so dependencies

state for the test suite. Any test that wants to use a fixture must explicitly accept it as an argument, so dependencies are always stated up front.

Fixtures can also make use of other fixtures, again by declaring them explicitly as dependencies. That means that, over time, your fixtures can become bulky and modular. Although the ability to insert fixtures into other fixtures provides enormous flexibility, it can also make managing dependencies more challenging as your test suite grows. Later in this tutorial, you'll learn more about fixtures and try a few techniques for handling these challenges.

Test Filtering

As your test suite grows, you may find that you want to run just a few tests on a feature and save the full suite for later. `pytest` provides a few ways of doing this:

- **Name-based filtering**: You can limit `pytest` to running only those tests whose fully qualified names match a particular expression. You can do this with the `-k` parameter.
- **Directory scoping**: By default, `pytest` will run only those tests that are in or under the current directory.
- **Test categorization**: `pytest` can include or exclude tests from particular categories that you define. You can do this with the `-m` parameter.

Test categorization in particular is a subtly powerful tool. `pytest` enables you to create **marks**, or custom labels, for any test you like. A test may have multiple labels, and you can use them for granular control over which tests to run. Later in this tutorial, you'll see an example of how `pytest` marks work and learn how to make use of them in a large test suite.

Test Parametrization

When you're testing functions that process data or perform generic transformations, you'll find yourself writing many similar tests. They may differ only in the input or output of the code being tested. This requires duplicating test code, and doing so can sometimes obscure the behavior you're trying to test.

`unittest` offers a way of collecting several tests into one, but they don't show up as individual tests in result reports. If one test fails and the rest pass, then the entire group will still return a single failing result. `pytest` offers its own solution in which each test can pass or fail independently. You'll see how to parametrize tests with `pytest` later in this tutorial.

Plugin-Based Architecture

One of the most beautiful features of `pytest` is its openness to customization and new features. Almost every piece of the program can be cracked open and changed. As a result, `pytest` users have developed a rich ecosystem of helpful plugins.

Although some `pytest` plugins focus on specific frameworks like Django, others are applicable to most test suites. You'll see details on some specific plugins later in this tutorial.

Fixtures: Managing State and Dependencies

`pytest` fixtures are a way of providing data, test doubles, or state setup to your tests. Fixtures are functions that can return a wide range of values. Each test that depends on a fixture must explicitly accept that fixture as an argument.

When to Create Fixtures

Imagine you're writing a function, `format_data_for_display()`, to process the data returned by an API endpoint. The data represents a list of people, each with a given name, family name, and job title. The function should output a list of strings that include each person's full name (their `given_name` followed by their `family_name`), a colon, and their `title`. To test this, you might write the following code:

Python

```
def format_data_for_display(people):
    ...  # Implement this!

def test_format_data_for_display():
```

Help

```
    people = [
        {
            "given_name": "Alfonsa",
            "family_name": "Ruiz",
            "title": "Senior Software Engineer",
        },
        {
            "given_name": "Sayid",
            "family_name": "Khan",
            "title": "Project Manager",
        },
    ]

    assert format_data_for_display(people) == [
        "Alfonsa Ruiz: Senior Software Engineer",
        "Sayid Khan: Project Manager",
    ]
```

Now suppose you need to write another function to transform the data into comma-separated values for use in Excel. The test would look awfully similar:

Python

```
def format_data_for_excel(people):
    ... # Implement this!

def test_format_data_for_excel():
    people = [
        {
            "given_name": "Alfonsa",
            "family_name": "Ruiz",
            "title": "Senior Software Engineer",
        },
        {
            "given_name": "Sayid",
            "family_name": "Khan",
            "title": "Project Manager",
        },
    ]

    assert format_data_for_excel(people) == """given,family,title
Alfonsa,Ruiz,Senior Software Engineer
Sayid,Khan,Project Manager
"""
```

If you find yourself writing several tests that all make use of the same underlying test data, then a fixture may be in your future. You can pull the repeated data into a single function decorated with `@pytest.fixture` to indicate that the function is a `pytest` fixture:

Python

```
import pytest

@pytest.fixture
def example_people_data():
    return [
        {
            "given_name": "Alfonsa",
            "family_name": "Ruiz",
            "title": "Senior Software Engineer",
        },
        {
            "given_name": "Sayid",
            "family_name": "Khan",
            "title": "Project Manager",
        },
    ]
```

Help

You can use the fixture by adding it as an argument to your tests. Its value will be the return value of the fixture function:

Python

```
def test_format_data_for_display(example_people_data):
    assert format_data_for_display(example_people_data) == [
        "Alfonsa Ruiz: Senior Software Engineer",
        "Sayid Khan: Project Manager",
    ]

def test_format_data_for_excel(example_people_data):
    assert format_data_for_excel(example_people_data) == """given,family,title
Alfonsa,Ruiz,Senior Software Engineer
Sayid,Khan,Project Manager
"""
```

Each test is now notably shorter but still has a clear path back to the data it depends on. Be sure to name your fixture something specific. That way, you can quickly determine if you want to use it when writing new tests in the future!

When to Avoid Fixtures

Fixtures are great for extracting data or objects that you use across multiple tests. They aren't always as good for tests that require slight variations in the data. Littering your test suite with fixtures is no better than littering it with plain data or objects. It might even be worse because of the added layer of indirection.

As with most abstractions, it takes some practice and thought to find the right level of fixture use.

Fixtures at Scale

As you extract more fixtures from your tests, you might see that some fixtures could benefit from further extraction. Fixtures are **modular**, so they can depend on other fixtures. You may find that fixtures in two separate test modules share a common dependency. What can you do in this case?

You can move fixtures from test modules into more general fixture-related modules. That way, you can import them back into any test modules that need them. This is a good approach when you find yourself using a fixture repeatedly throughout your project.

`pytest` looks for `conftest.py` modules throughout the directory structure. Each `conftest.py` provides configuration for the file tree `pytest` finds it in. You can use any fixtures that are defined in a particular `conftest.py` throughout the file's parent directory and in any subdirectories. This is a great place to put your most widely used fixtures.

Another interesting use case for fixtures is in guarding access to resources. Imagine that you've written a test suite for code that deals with API calls. You want to ensure that the test suite doesn't make any real network calls, even if a test accidentally executes the real network call code. `pytest` provides a `monkeypatch` fixture to replace values and behaviors, which you can use to great effect:

Python

```
# conftest.py

import pytest
import requests

@pytest.fixture(autouse=True)
def disable_network_calls(monkeypatch):
    def stunted_get():
        raise RuntimeError("Network access not allowed during testing!")
    monkeypatch.setattr(requests, "get", lambda *args, **kwargs: stunted_get())
```

Help

By placing `disable_network_calls()` in `conftest.py` and adding the `autouse=True` option, you ensure that network calls will be disabled in every test across the suite. Any test that executes code calling `requests.get()` will raise a

`RuntimeError` indicating that an unexpected network call would have occurred.

Marks: Categorizing Tests

In any large test suite, some of the tests will inevitably be slow. They might test timeout behavior, for example, or they might exercise a broad area of the code. Whatever the reason, it would be nice to avoid running *all* the slow tests when you're trying to iterate quickly on a new feature.

`pytest` enables you to define categories for your tests and provides options for including or excluding categories when you run your suite. You can mark a test with any number of categories.

Marking tests is useful for categorizing tests by subsystem or dependencies. If some of your tests require access to a database, for example, then you could create a `@pytest.mark.database_access` mark for them.

> **Pro tip**: Because you can give your marks any name you want, it can be easy to mistype or misremember the name of a mark. `pytest` will warn you about marks that it doesn't recognize.
>
> The `--strict-markers` flag to the `pytest` command ensures that all marks in your tests are registered in your `pytest` configuration. It will prevent you from running your tests until you register any unknown marks.
>
> For more information on registering marks, check out the `pytest` documentation.

When the time comes to run your tests, you can still run them all by default with the `pytest` command. If you'd like to run only those tests that require database access, then you can use `pytest -m database_access`. To run all tests *except* those that require database access, you can use `pytest -m "not database_access"`. You can even use an `autouse` fixture to limit database access to those tests marked with `database_access`.

Some plugins expand on the functionality of marks by guarding access to resources. The `pytest-django` plugin provides a `django_db` mark. Any tests without this mark that try to access the database will fail. The first test that tries to access the database will trigger the creation of Django's test database.

The requirement that you add the `django_db` mark nudges you toward stating your dependencies explicitly. That's the `pytest` philosophy, after all! It also means that you can run tests that don't rely on the database much more quickly, because `pytest -m "not django_db"` will prevent the test from triggering database creation. The time savings really add up, especially if you're diligent about running your tests frequently.

`pytest` provides a few marks out of the box:

- **`skip`** skips a test unconditionally.
- **`skipif`** skips a test if the expression passed to it evaluates to `True`.
- **`xfail`** indicates that a test is expected to fail, so if the test *does* fail, the overall suite can still result in a passing status.
- **`parametrize`** (note the spelling) creates multiple variants of a test with different values as arguments. You'll learn more about this mark shortly.

You can see a list of all the marks `pytest` knows about by running `pytest --markers`.

Parametrization: Combining Tests

You saw earlier in this tutorial how `pytest` fixtures can be used to reduce code duplication by extracting common dependencies. Fixtures aren't quite as useful when you have several tests with slightly different inputs and expected outputs. In these cases, you can **parametrize** a single test definition, and `pytest` will create variants of the test for yo with the parameters you specify.

Imagine you've written a function to tell if a string is a palindrome. An initial set of tests could look like this:

Python

```
def test_is_palindrome_empty_string():
    assert is_palindrome("")

def test_is_palindrome_single_character():
    assert is_palindrome("a")

def test_is_palindrome_mixed_casing():
    assert is_palindrome("Bob")

def test_is_palindrome_with_spaces():
    assert is_palindrome("Never odd or even")

def test_is_palindrome_with_punctuation():
    assert is_palindrome("Do geese see God?")

def test_is_palindrome_not_palindrome():
    assert not is_palindrome("abc")

def test_is_palindrome_not_quite():
    assert not is_palindrome("abab")
```

All of these tests except the last two have the same shape:

Python

```
def test_is_palindrome_<in some situation>():
    assert is_palindrome("<some string>")
```

You can use `@pytest.mark.parametrize()` to fill in this shape with different values, reducing your test code significantly:

Python

```
@pytest.mark.parametrize("palindrome", [
    "",
    "a",
    "Bob",
    "Never odd or even",
    "Do geese see God?",
])
def test_is_palindrome(palindrome):
    assert is_palindrome(palindrome)

@pytest.mark.parametrize("non_palindrome", [
    "abc",
    "abab",
])
def test_is_palindrome_not_palindrome(non_palindrome):
    assert not is_palindrome(non_palindrome)
```

The first argument to `parametrize()` is a comma-delimited string of parameter names. The second argument is a list of either tuples or single values that represent the parameter value(s). You could take your parametrization a step further to combine all your tests into one:

Python

```
@pytest.mark.parametrize("maybe_palindrome, expected_result", [
    ("", True),
    ("a", True),
    ("Bob", True),
    ("Never odd or even", True),
    ("Do geese see God?", True),
    ("abc", False),
    ("abab", False),
])
```

```
def test_is_palindrome(maybe_palindrome, expected_result):
    assert is_palindrome(maybe_palindrome) == expected_result
```

Even though this shortened your code, it's important to note that in this case, it didn't do much to clarify your test code. Use parametrization to separate the test data from the test behavior so that it's clear what the test is testing!

Durations Reports: Fighting Slow Tests

Each time you switch contexts from implementation code to test code, you incur some overhead. If your tests are slow to begin with, then overhead can cause friction and frustration.

You read earlier about using marks to filter out slow tests when you run your suite. If you want to improve the speed of your tests, then it's useful to know which tests might offer the biggest improvements. `pytest` can automatically record test durations for you and report the top offenders.

Use the `--durations` option to the `pytest` command to include a duration report in your test results. `--durations` expects an integer value `n` and will report the slowest `n` number of tests. The output will follow your test results:

Shell

```
$ pytest --durations=3
3.03s call     test_code.py::test_request_read_timeout
1.07s call     test_code.py::test_request_connection_timeout
0.57s call     test_code.py::test_database_read
======================== 7 passed in 10.06s ==============================
```

Each test that shows up in the durations report is a good candidate to speed up because it takes an above-average amount of the total testing time.

Be aware that some tests may have an invisible setup overhead. You read earlier about how the first test marked with `django_db` will trigger the creation of the Django test database. The `durations` report reflects the time it takes to set up the database in the test that triggered the database creation, which can be misleading.

Useful `pytest` Plugins

You learned about a few valuable `pytest` plugins earlier in this tutorial. You can explore those and a few others in more depth below.

`pytest-randomly`

`pytest-randomly` does something seemingly simple but with valuable effect: It forces your tests to run in a random order. `pytest` always collects all the tests it can find before running them, so `pytest-randomly` shuffles that list of tests just before execution.

This is a great way to uncover tests that depend on running in a specific order, which means they have a **stateful dependency** on some other test. If you built your test suite from scratch in `pytest`, then this isn't very likely. It's more likely to happen in test suites that you migrate to `pytest`.

The plugin will print a seed value in the configuration description. You can use that value to run the tests in the same order as you try to fix the issue.

`pytest-cov`

If you measure how well your tests cover your implementation code, you likely use the coverage package. `pytest-cov` integrates coverage, so you can run `pytest --cov` to see the test coverage report.

pytest-django

pytest-django provides a handful of useful fixtures and marks for dealing with Django tests. You saw the `django_db` mark earlier in this tutorial, and the `rf` fixture provides direct access to an instance of Django's `RequestFactory`. The `settings` fixture provides a quick way to set or override Django settings. This is a great boost to your Django testing productivity!

If you're interested in learning more about using `pytest` with Django, then check out How to Provide Test Fixtures for Django Models in Pytest.

pytest-bdd

`pytest` can be used to run tests that fall outside the traditional scope of unit testing. Behavior-driven development (BDD) encourages writing plain-language descriptions of likely user actions and expectations, which you can then use to determine whether to implement a given feature. pytest-bdd helps you use Gherkin to write feature tests for your code.

You can see which other plugins are available for `pytest` with this extensive list of third-party plugins.

Conclusion

`pytest` offers a core set of productivity features to filter and optimize your tests along with a flexible plugin system that extends its value even further. Whether you have a huge legacy `unittest` suite or you're starting a new project from scratch, `pytest` has something to offer you.

In this tutorial, you learned how to use:

- **Fixtures** for handling test dependencies, state, and reusable functionality
- **Marks** for categorizing tests and limiting access to external resources
- **Parametrization** for reducing duplicated code between tests
- **Durations** to identify your slowest tests
- **Plugins** for integrating with other frameworks and testing tools

Install `pytest` and give it a try. You'll be glad you did. Happy testing!