Extracted from:

Python Testing with pytest, Second Edition

Simple, Rapid, Effective, and Scalable

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Python Testing with pytest

Second Edition

Simple, Rapid, Effective, and Scalable

Brian Okken

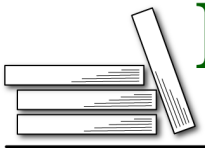
edited by Katharine Dvorak

Python Testing with pytest, Second Edition

Simple, Rapid, Effective, and Scalable

Brian Okken

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit *https://pragprog.com*.

The team that produced this book includes:

CEO: Dave Rankin
COO: Janet Furlow
Managing Editor: Tammy Coron
Development Editor: Katharine Dvorak
Copy Editor: Karen Galle
Indexing: Potomac Indexing, LLC
Layout: Gilson Graphics
Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

ISBN-13: 978-1-68050-860-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—February 2022

CHAPTER 3

pytest Fixtures

Now that you've used pytest to write and run test functions, let's turn our attention to test helper functions called *fixtures*, which are essential to structuring test code for almost any non-trivial software system. Fixtures are functions that are run by pytest before (and sometimes after) the actual test functions. The code in the fixture can do whatever you want it to. You can use fixtures to get a data set for the tests to work on. You can use fixtures to get a system into a known state before running a test. Fixtures are also used to get data ready for multiple tests.

In this chapter, you'll learn how to create fixtures and learn how to work with them. You'll learn how to structure fixtures to hold both setup and teardown code. You'll use scope to allow fixtures to run once over many tests, and learn how tests can use multiple fixtures. You'll also learn how to trace code execution through fixtures and test code.

But first, before you learn the ins and outs of fixtures and use them to help test Cards, let's look at a small example fixture and how fixtures and test functions are connected.

Getting Started with Fixtures

Here's a simple fixture that returns a number:

ch3/test_fixtures.py

```
import pytest

@pytest.fixture()
def some_data():
    """Return answer to ultimate question."""
    return 42
```

```
def test_some_data(some_data):
    """Use fixture return value in a test."""
    assert some_data == 42
```

The @pytest.fixture() decorator is used to tell pytest that a function is a fixture. When you include the fixture name in the parameter list of a test function, pytest knows to run it before running the test. Fixtures can do work, and can also return data to the test function.

You don't need to have a complete understanding of Python decorators to use the decorators included with pytest. pytest uses decorators to add functionality and features to other functions. In this case, pytest.fixture() is decorating the some_data() function. The test, test_some_data(), has the name of the fixture, some_data, as a parameter. pytest will see this and look for a fixture with this name.

The term *fixture* has many meanings in the programming and test community, and even in the Python community. I use "fixture," "fixture function," and "fixture method" interchangeably to refer to the @pytest.fixture() decorated functions discussed in this chapter. Fixture can also be used to refer to the resource that is being set up by the fixture functions. Fixture functions often set up or retrieve some data that the test can work with. Sometimes this data is considered a fixture. For example, the Django community often uses *fixture* to mean some initial data that gets loaded into a database at the start of an application.

Regardless of other meanings, in pytest and in this book, test fixtures refer to the mechanism pytest provides to allow the separation of "getting ready for" and "cleaning up after" code from your test functions.

pytest treats exceptions differently during fixtures compared to during a test function. An exception (or assert failure or call to pytest.fail()) that happens during the test code proper results in a "Fail" result. However, during a fixture, the test function is reported as "Error." This distinction is helpful when debugging why a test didn't pass. If a test results in "Fail," the failure is somewhere in the test function (or something the function called). If a test results in "Error," the failure is somewhere in a fixture.

pytest fixtures are one of the unique core features that make pytest stand out above other test frameworks, and are the reason why many people switch to and stay with pytest. There are a lot of features and nuances about fixtures. Once you get a good mental model of how they work, they will seem easy to you. However, you have to play with them a while to get there, so let's do that next.

Using Fixtures for Setup and Teardown

Fixtures are going to help us a lot with testing the Cards application. The Cards application is designed with an API that does most of the work and logic, and a thin CLI. Especially because the user interface is rather thin on logic, focusing most of our testing on the API will give us the most bang for our buck. The Cards application also uses a database, and dealing with the database is where fixtures are going to help out a lot.

Make Sure Cards Is Installed

Examples in this chapter require having the Cards application installed. If you haven't already installed the Cards application, be sure to install it with `cd code; pip install ./cards_proj`. See Installing the Sample Application, on page ? for more information.

Let's start by writing some tests for the `count()` method that supports the `count` functionality. As a reminder, let's play with `count` on the command line:

```
$ cards count
0
$ cards add first item
$ cards add second item
$ cards count
2
```

An initial test, checking that the count starts at zero, might look like this:

ch3/test_count_initial.py

```
from pathlib import Path
from tempfile import TemporaryDirectory
import cards

def test_empty():
    with TemporaryDirectory() as db_dir:
        db_path = Path(db_dir)
        db = cards.CardsDB(db_path)

        count = db.count()
        db.close()

        assert count == 0
```

In order to call `count()`, we need a database object, which we get by calling `cards.CardsDB(db_path)`. The `cards.CardsDB()` function is a constructor; it returns a `CardsDB` object. The `db_path` parameter needs to be a `pathlib.Path` object that points to the database directory. The `pathlib` module was introduced in Python 3.4

and pathlib.Path[1] objects are the standard way to represent file system paths. For testing, a temporary directory works, which we get from tempfile.TemporaryDirectory(). There are other ways to get all of this done, but this works for now.

This test function really isn't too painful. It's only a few lines of code. Let's look at the problems anyway. There is code to get the database set up before we call count() that isn't really what we want to test. There is the call to db.close() before the assert statement. It would seem better to place this at the end of the function, but we have to call it before assert, because if the assert statement fails, it won't be called.

These problems are resolved with a pytest fixture:

ch3/test_count.py

```
import pytest

@pytest.fixture()
def cards_db():
    with TemporaryDirectory() as db_dir:
        db_path = Path(db_dir)
        db = cards.CardsDB(db_path)
        yield db
        db.close()

def test_empty(cards_db):
    assert cards_db.count() == 0
```

Right off the bat we can see that the test function itself is way easier to read, as we've pushed all the database initialization into a fixture called cards_db.

The cards_db fixture is "setting up" for the test by getting the database ready. It's then yield-ing the database object. That's when the test gets to run. And then after the test runs, it closes the database.

Fixture functions run before the tests that use them. If there is a yield in the function, it stops there, passes control to the tests, and picks up on the next line after the tests are done. The code above the yield is "setup" and the code after yield is "teardown." The code after the yield, the teardown, is guaranteed to run regardless of what happens during the tests.

In our example, the yield happens within a context manager with block for the temporary directory. That directory stays around while the fixture is in use and the tests run. After the test is done, control passes back to the fixture, the db.close() can run, and then the with block can complete and clean up the directory.

1. https://docs.python.org/3/library/pathlib.html#basic-use

Remember: pytest looks at the specific name of the arguments to our test and then looks for a fixture with the same name. We never call fixture functions directly. pytest does that for us.

You can use fixtures in multiple tests. Here's another one:

ch3/test_count.py

```
def test_two(cards_db):
    cards_db.add_card(cards.Card("first"))
    cards_db.add_card(cards.Card("second"))
    assert cards_db.count() == 2
```

test_two() uses the same cards_db fixture. This time, we take the empty database and add two cards before checking the count. We can now use cards_db for any test that needs a configured database to run. The individual tests, such as test_empty() and test_two() can be kept smaller and focus on what we are testing, and not the setup and teardown bits.

The fixture and test function are separate functions. Carefully naming your fixtures to reflect the work being done in the fixture or the object returned from the fixture, or both, will help with readability.

While writing and debugging test functions, it's frequently helpful to visualize when the setup and teardown portions of fixtures run with respect the tests using them. The next section describes --setup-show to help with this visualization.