Extracted from:

Python Testing with pytest

Simple, Rapid, Effective, and Scalable

The Pragmatic Bookshelf

Raleigh, North Carolina

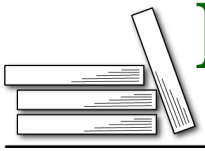
Python Testing with pytest

Simple, Rapid, Effective, and Scalable

Brian Okken

The Pragmatic Bookshelf

Raleigh, North Carolina



Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Publisher: Andy Hunt
VP of Operations: Janet Furlow
Development Editor: Katharine Dvorak
Indexing: Potomac Indexing, LLC
Copy Editor: Nicole Abramowitz
Layout: Gilson Graphics

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

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

Printed in the United States of America.
ISBN-13: 978-1-68050-240-4
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—September 2017

Now that you've seen the basics of pytest, let's turn our attention to 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.

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.

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. Naming is significant in pytest. pytest will look in the module of the test for a fixture of that name. It will also look in conftest.py files if it doesn't find it in this file.

Before we start our exploration of fixtures (and the conftest.py file), I need to address the fact that 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 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. However, fixtures in pytest are different than fixtures in Django and different than the setup and teardown procedures found in unittest and nose. 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 get started.

Sharing Fixtures Through conftest.py

You can put fixtures into individual test files, but to share fixtures among multiple test files, you need to use a conftest.py file somewhere centrally located for all of the tests. For the Tasks project, all of the fixtures will be in tasks_proj/tests/conftest.py.

From there, the fixtures can be shared by any test. You can put fixtures in individual test files if you want the fixture to only be used by tests in that file. Likewise, you can have other conftest.py files in subdirectories of the top tests directory. If you do, fixtures defined in these lower-level conftest.py files will be available to tests in that directory and subdirectories. So far, however, the fixtures in the Tasks project are intended to be available to any test. Therefore, putting all of our fixtures in the conftest.py file at the test root, tasks_proj/tests, makes the most sense.

Although conftest.py is a Python module, it should not be imported by test files. Don't import conftest from anywhere. The conftest.py file gets read by pytest, and is considered a local *plugin*, which will make sense once we start talking about plugins in Chapter 5, *Plugins*, on page ?. For now, think of tests/conftest.py as a place where we can put fixtures used by all tests under the tests directory.

Next, let's rework some our tests for tasks_proj to properly use fixtures.

Using Fixtures for Setup and Teardown

Most of the tests in the Tasks project will assume that the Tasks database is already set up and running and ready. And we should clean things up at the end if there is any cleanup needed. And maybe also disconnect from the database. Luckily, most of this is taken care of within the tasks code with tasks.start_tasks_db(<directory to store db>, 'tiny' or 'mongo') and tasks.stop_tasks_db(); we just need to call them at the right time, and we need a temporary directory.

Fortunately, pytest includes a cool fixture called tmpdir that we can use for testing and don't have to worry about cleaning up. It's not magic, just good

coding by the pytest folks. (Don't worry; we look at tmpdir and it's session-scoped relative tmpdir_factory in more depth in *Using tmpdir and tmpdir_factory*, on page ?.)

Given those pieces, this fixture works nicely:

ch3/a/tasks_proj/tests/conftest.py

```
import pytest
import tasks
from tasks import Task

@pytest.fixture()
def tasks_db(tmpdir):
    """Connect to db before tests, disconnect after."""
    # Setup : start db
    tasks.start_tasks_db(str(tmpdir), 'tiny')

    yield  # this is where the testing happens

    # Teardown : stop db
    tasks.stop_tasks_db()
```

The value of tmpdir isn't a string—it's an object that represents a directory. However, it implements __str__, so we can use str() to get a string to pass to start_tasks_db(). We're still using 'tiny' for TinyDB, for now.

A fixture function runs before the tests that use it. However, 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. Therefore, think of the code above the yield as "setup" and the code after yield as "teardown." The code after the yield, the "teardown," is guaranteed to run regardless of what happens during the tests. We're not returning any data with the yield in this fixture. But you can.

Let's change one of our tasks.add() tests to use this fixture:

ch3/a/tasks_proj/tests/func/test_add.py

```
import pytest
import tasks
from tasks import Task

def test_add_returns_valid_id(tasks_db):
    """tasks.add(<valid task>) should return an integer."""
    # GIVEN an initialized tasks db
    # WHEN a new task is added
    # THEN returned task_id is of type int
    new_task = Task('do something')
    task_id = tasks.add(new_task)
    assert isinstance(task_id, int)
```

The main change here is that the extra fixture in the file has been removed, and we've added `tasks_db` to the parameter list of the test. I like to structure tests in a GIVEN/WHEN/THEN format using comments, especially when it isn't obvious from the code what's going on. I think it's helpful in this case. Hopefully, `GIVEN an initialized tasks db` helps to clarify why `tasks_db` is used as a fixture for the test.