pytest Fixtures

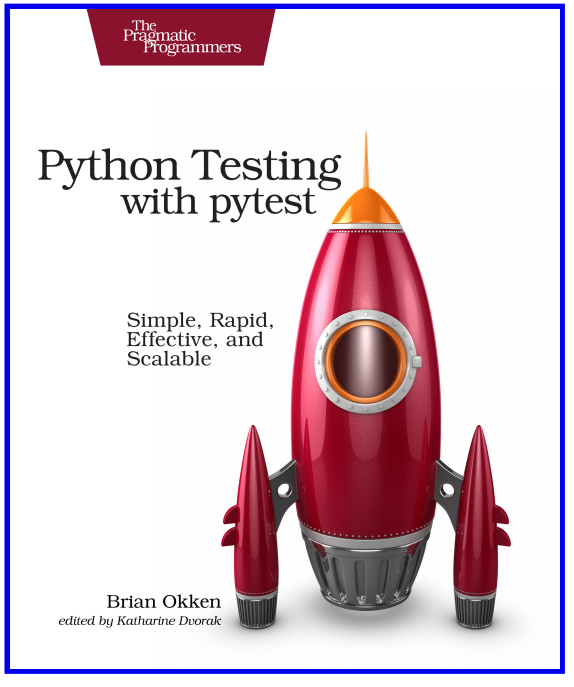
Brian Okken

@brianokken

Code and slides

github.com/okken/talks

Brian Okken



Podcasts

TEST & CODE

PYTHON BYTES
Python headlines delivered directly to your earbuds

Also

- Lead Engineer at R&S
- Mostly Comms: Currently WLAN RF Measurements
- Also SpecAns, Scopes, Satellite Systems

@brianokken

Why pytest?

- simple asserts
- fixtures
  - setup and teardown
  - pass data to tests
  - can be shared across a project
  - support parametrization
  - can even be shared as a plugin for multiple projects
- parametrization
  - tons-o-test-cases with a single test
- plugins
  - extend functionality
  - integrate with other tools
  - share code
- markers
  - one way to easily run subsets of tests
  - can work with fixtures/plugins to extend pytest functionality

Why pytest?

- simple asserts
- **fixtures**
  - **setup and teardown**
  - **pass data to tests**
  - **can be shared across a project**
  - support parametrization
  - can even be shared as a plugin for multiple projects
- parametrization
  - tons-o-test-cases with a single test
- plugins
  - extend functionality
  - integrate with other tools
  - share code
- markers
  - one way to easily run subsets of tests
  - can work with fixtures/plugins to extend pytest functionality

A Common Structure for Tests

- Setup
- Action
- Check outcome of action
- Teardown

If you're familiar with:

- Given, When, Then
- Arrange, Act, Assert

The first three steps are the same.
The teardown stage is implied in those models, I guess.

Without Fixtures

```
# test_no_fixtures.py

def test_count_empty():
    # Setup
    with TemporaryDirectory() as db_dir:
        db = some_db.DB(db_dir, "my_db")

        # Action / Check
        assert db.count() == 0

        # Teardown
        db.close()
```

xUnit Solution

```
# test_xunit.py

def setup_function(function):
    global _dir, _db
    _dir = TemporaryDirectory()
    _db = some_db.DB(_dir.name, "my_db")

def test_count_empty():
    assert _db.count() == 0

def teardown_function(function):
    _db.close()
    _dir.cleanup()
```

```
# test_no_fixtures.py

def test_count_empty():
    with TemporaryDirectory() as db_dir:
        db = some_db.DB(db_dir, "my_db")

        assert db.count() == 0

        db.close()
```

xUnit Solution

```
# test_xunit.py

def setup_function(function):
    global _dir, _db
    _dir = TemporaryDirectory()
    _db = some_db.DB(_dir.name, "my_db")

def test_count_empty():
    assert _db.count() == 0

def teardown_function(function):
    _db.close()
    _dir.cleanup()
```

the good:

- The test itself is simpler and easier to read
- Failing test can't stop teardown
- Many tests can share the same setup/teardown

xUnit Solution

```
# test_xunit.py

def setup_function(function):
    global _dir, _db
    _dir = TemporaryDirectory()
    _db = some_db.DB(_dir.name, "my_db")

def test_count_empty():
    assert _db.count() == 0

def teardown_function(function):
    _db.close()
    _dir.cleanup()
```

the bad:

- Can't use a context managers across setup/teardown.
- Multiple resources can get clunky.
- Failures during setup gets complicated with multiple resources.
- Is readability/maintainability better?

xUnit Solution

```
# test_xunit.py

def setup_function(function):
    global _dir, _db
    _dir = TemporaryDirectory()
    _db = some_db.DB(_dir.name, "my_db")

def test_count_empty():
    assert _db.count() == 0

def teardown_function(function):
    _db.close()
    _dir.cleanup()
```

```
# test_no_fixtures.py

def test_count_empty():
    with TemporaryDirectory() as db_dir:
        db = some_db.DB(db_dir, "my_db")

        assert db.count() == 0

        db.close()
```

unittest, if your curious

```
# test_unittest.py

class TestCount(unittest.TestCase):

    def setUp(self):
        global _dir, _db
        self._dir = TemporaryDirectory()
        self._db = some_db.DB(self._dir.name, "my_db")

    def test_count_empty(self):
        self.assertEqual(self._db.count(), 0)

    def tearDown(self):
        self._db.close()
        self._dir.cleanup()
```

- Previous example was pytest version of xUnit fixtures.
- pytest can run both.
- Mostly the same, just the class and all the self everywhere.
- assertEqual instead of assert

pytest Fixture Solution

```
# test_pytest.py

@pytest.fixture()
def db():
    with TemporaryDirectory() as db_dir:
        _db = some_db.DB(db_dir, "my_db")
        yield _db  # yield separates setup & teardown
        _db.close()

def test_count_empty(db):
    assert db.count() == 0
```

the good:

- Failing test can't stop teardown
- Many tests can share the same setup/teardown
- Setup/Teardown in same function
- Can use a context managers across setup/teardown
- Pass data to test with return or yield
- Multiple resources are no problem, use multiple fixtures, or layers of fixtures
- Multipe fixtures also solves the expensive setup problem

Multiple Fixture Scopes

Expensive setup can be solved by splitting into two fixtures

```
# test_scope.py

@pytest.fixture(scope="session")
def db_session():
    with TemporaryDirectory() as db_dir:
        _db = some_db.DB(db_dir, "my_db")
        yield _db
        _db.close()

@pytest.fixture()
def db(db_session):
    db_session.delete_all()
    return db_session

def test_count_empty(db):
    assert db.count() == 0
```

- test_count_empty() depends on db which depends on db_session
- Connect to database once per test session
- Clean it out for each test

Example of Fixture Levels

Why fixture levels convinced me to use pytest

```
@pytest.fixture(scope="session")
def device_session():
    _dev = connect_to_device()
    return _dev

@pytest.fixture(scope="function")
def device(device_session):
    _dev = device_session
    _dev.write('*RST')
    _dev.query('*OPC?')
    return _dev

def test_id(device):
    id = device.query('*IDN?')
    assert id == "Some Expected Instrument Id"
```

We can add checks without changing tests

```
@pytest.fixture(scope="session")
def device_session():
    _dev = connect_to_device()
    return _dev

@pytest.fixture(scope="function")
def check_device_logs(device_session):
    _dev = device_session
    yield _dev
    _dev.assert_clean_error_logs()

@pytest.fixture(scope="function")
def device(check_device_logs):
    _dev = check_device_logs
    _dev.write('*RST')
    _dev.query('*OPC?')
    return _dev


def test_id(device):
    id = device.query('*IDN?')
    assert id == "Some Expected Instrument Id"
```

Builtin fixtures

```
# test_builtin.py

@pytest.fixture(scope="session")
def db_session(tmp_path_factory):
    path = tmp_path_factory.mktemp("db_dir")
    _db = some_db.DB(path, "my_db")
    yield _db
    _db.close()

@pytest.fixture()
def db(db_session):
    db_session.delete_all()
    return db_session

def test_count_empty(db):
    assert db.count() == 0
```

- Temp files/directories is so common, it's a builtin for pytest
- Other cool builtins like capsys and monkeypatch available.
- Many more pre-built fixtures available with pytest plugins on pypi.org

Sharing is Caring

We can share fixtures with other test files.

```
# tests/conftest.py

@pytest.fixture(scope="session")
def db_session(tmp_path_factory):
    """Session db connection"""
    path = tmp_path_factory.mktemp("db_dir")
    _db = some_db.DB(path, "my_db")
    yield _db
    _db.close()

@pytest.fixture()
def db(db_session):
    """Clean db per test"""
    db_session.delete_all()
    return db_session
```

- Fixtures in conftest.py can be used by any test in this directory or any subdirectory with no import needed.

Focused Test Files

Now my test files can be small and focused, if I want.

```
# tests/test_count.py

def test_count_empty(db):
    assert db.count() == 0

def test_count_one(db):
    item = {"foo": [1, 2, 3]}
    db.create(item)
    assert db.count() == 1
```

```
# tests/test_update.py

def test_update(db):
    id = db.create({"foo": [1, 2, 3], "bar": [4, 5, 6]})

    db.update(id, {"bar": "baz"})

    expected = {"foo": [1, 2, 3], "bar": "baz"}
    assert db.read(id) == expected
```

Tracing Execution

To help you get your head around the fixture control flow

```
$ pytest --setup-show test_scope.py
============================= test session starts ==============================
collected 1 item

test_scope.py
SETUP    S db_session
        SETUP    F db (fixtures used: db_session)
        test_scope.py::test_count_empty (fixtures used: db, db_session).
        TEARDOWN F db
TEARDOWN S db_session

=========================== 1 passed in 0.01s ==============================
```

- See which fixtures are run, when with pytest --setup-show

Finding Fixtures in Your Own Code

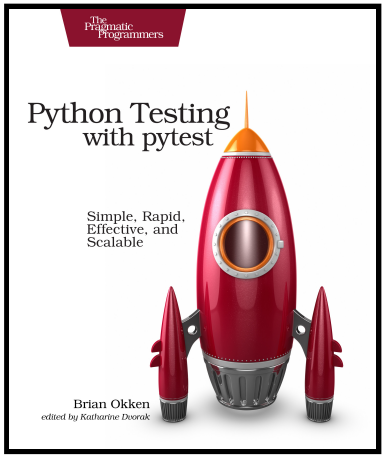
Where's that fixture defined?

```
$ pytest --fixtures -v
...
< all the built in ones and those from plugins >
...
---------------------- fixtures defined from conftest ----------------------
db -- conftest.py:15
    Clean db per test

db_session [session scope] -- conftest.py:6
    Session db connection

--------------------- fixtures defined from test_crud ----------------------
some_dict -- test_crud.py:5
    test_crud.py:5: no docstring available
...
```

- Find where they are defined with pytest --fixtures -v
  - Gives file names and line numbers
  - Even docstrings, if provided

Learn More about pytest

- Python Testing with pytest, pytestbook.com
  - The fastest way to get super productive with pytest
  - Also on Medium
  - Look for it here: medium.com/pragmatic-programmers
- Corporate Training:
  - testandcode.com/training
- Test & Code Podcast
  - testandcode.com
- Slack Community
  - testandcode.com/slack
- This code, and slides
  - github.com/okken/talks under 2021/Aberdeen/fixtures.
- Oh yeah
  - there's also pytest.org



@brianokken