Effective testing with Pytest

@hectorcanto_dev

slideshare.net/HectorCanto

github.com/hectorcanto/pytest-samples

Summary

We will focus in **Python testing with Pytest**, but many things apply to any test framework in any language

Why pytest

IMHO pytest is a top-grade framework, updated, very well documented, and compatible with many tools.

Why we need testing effectively

Testing takes a good part of our time working on Software

- The best we test, the better software we make
- The most effective we are, more time to do other stuff

Testing is very important

- Guarantees all our code works when adding new features
- Documents it with examples and context
- Debugging environment
- **Enables refactoring**

Article: Tips and tricks for unit tests

Effectivity means

- Running test as fast as possibly
- Developing and maintaining code
- Readable for everyone, especially newcomers

How we achieve effectivity

- Using Pytest to its **full potential**
- Understanding artifacts: mocks, fixtures ...
- Applying best practices
- Adding good libs and plugins
- Planning and common sense

Test phases

A test can be divided in 3 phases know as the triple A (AAA)

- Arrange
- Act
- Assert

We will have points on every one of them

Launching the suite

- Pytest options
- Markers
- Test selection

Launch only what you need at every moment

Pytest options

```
pytest --help
pytest tests/folder/test_file.py::test_name
pytest -m smoke
pytest -k users
```

We can filter tests in several ways

Suite: repeat failed tests

```
pytest --fail-first
pytest --last-failed
pytest --failed-first
```

We will give priority to failed ones

Suite: specific tests

```
def test_users_creation():
    ...

def test_users_update():
    ...
```

Naming is key

```
pytest -k users
```

Suite: markers

```
@pytest.mark.slow
@pytest.mark.current
@pytest.mark.skip(reason="whatever")
@pytest.mark.xfail
def test_this():
    ...
```

```
pytest -m current -s  -v
pytest -m "slow and not integration"
pytest -m "smoke and unit"
```

Useful markers: smoke, unit, integration, current, slow

Suite: global markers

```
# Per module
pytestmark = pytest.mark.auth
pytestmark = [pytest.mark.deletion, pytest.mark.api]

# Per class
class TestClass:
    pytestmark = pytest.mark.special
```

It is easy to mark groups of tests

Suite: Folder structure

```
tests
├── smoke/
├── unit/
├── integration/
├── api/
├
├── fixtures/
├── factories/
├── conftest.py
├── aux.py
└── __init__.py
```

Suite: automatic markers

```python
def pytest_collection_modifyitems(items):
    for item in items:
        if "/smoke/" in str(item.module):
            item.add_marker("smoke")
```

Auto-mark tests by folder at `tests/conftest.py`

Launching the suite: test types

- Smoke tests: very simple ones. Run first
- Unit test: small and independent. Run often
- Integration: they need something external. Run sometimes
- API: outside the box. Run sometimes

Suite: ordering

```
@pytest.mark.first
@pytest.mark.second
@pytest.mark.last
```

Ordering is useful for smoke tests and on CI pipeline

Randomization helps finding lateral effects

```
Plugins: pytest-ordering, pytest-randomly
```

Suite: environment

```
# setup.cfg
[tool:pytest]
env =
    PYTHONBREAKPOINT=ipdb.set_trace
    APP_ENVIRONMENT=test
    CACHE=memory
    DEBUG=1
    VAR=value
```

We keep local and test envs apart

```
Plugin: pytest-env
```

Suite: environment 2

```
def test_with_different_env_vars(monkeypatch):
    monkeypatch.setenv("CACHE", "nocache")
    monkeypatch.delenv("VAR")
```

```
Fixture: *monkeypatch*
```

Setup or Arrange

- Parametrization
- Fixtures
- Factories

Setup: Parametrization

```python
@pytest.mark.parametrize("entry, expected", (
    (1, True),
    (2, False),
    (None, False)
    ("hello", False)
)
def test_something(entry, expected: bool):
    result = function_under_test(entry)
    assert result == expected
```

Instead of making a test for each input

<ake one test for all inputs

Fixtures

In general, fixtures are the artifacts all around a test

- Synthetic data used as input
- The system in a specific state
- Active elements that interact with the SUT

Pytest fixtures

```
@pytest.fixture
def example_fixture()
    now = datetime.utcnow()
    do_setup(now)
    return now
    do_teardown()
```

In Pytest, fixtures are created with an especial decorator

There might be fixtures with setup, teardown or both

Fixture example

```python
@pytest.fixture
def load_data(db_client):
    my_user = User(name="Hector", last_name="Canto")
    db_client.add(my_user)
    yield my_user  # usable as parameter
    db_client.delete(my_user)
```

Fixture usage

```python
@pytest.mark.usefixtures("load_data")
def test_with_fixtures(example_fixture):
    result, error = system_under_tests(example_fixture)
    assert result
    assert not error
```

Place your fixtures on any `conftest.py`

No need to import your fixtures

Reuse fixtures

```
@pytest.fixture(scope="session")
def test_settings():
     yield get_settings(test=True)
```

Scopes: session, module, class or function

Automatic fixtures

```python
@pytest.fixture(autouse=True)
def clean_db(db_client):
    yield db_client
    for table in db_client.tables:
        db_client.truncate()
```

It always runs

It depends on another fixture

Data fixtures

```
Create your data fixture programmatically
```

- Import constants
- Generation functions
- Factories

No raw json, text files ...

Setup: Factory

```
from factory import StubFactory

class ObjectFactory(StubFactory):
    name: str = "value"

my_ojb = ObjectFactory()
assert my_obj.name == "value"

other = ObjectFactory(name="another value")
assert other.name == "another value"
```

```
Libary & plugin: factoryboy, pytest-factoryboy
```

Factory & Faker

```
from factoryboy import Faker, SelfAttribute
from factory.fuzzy import FuzzyInteger

class ObjectFactory(StubFactory):
    name: str = Faker("first_name_male")
    last_name: str = Faker("last_name")
    full_name: str = SelfAttribute(lambda self: f"{self.name} {se
    phone: str = Faker("phone_number", local="en_GB")
    money: int = FuzzyInteger(1, 1000)
```

Generate multiple values in one go

```
Library: Faker
```

Model Factory

```
from factory import alchemy, RelatedFactory

class UserFactory(alchemy.SQLAlchemyModelFactory):
    class Meta:
        model = User
        sqlalchemy_session = Session

    id = LazyFunction(lambda: randint(1, 1_000_000))
    location = RelatedFactory(LocationFactory)
    created_at = Faker(
        "unix_time",
        start_datetime=datetime(2015, 1, 1),
        end_datetime=datetime(2019, 12, 31)
    )
```

Can be used with Django, SQLAlchemy and Pymongo

Batches & Dict Factory

```
factory.build(dict, FACTORY_CLASS=UserFactory)
UserFactory.create_batch(5)
UserFactory.build_batch(5)  # created but not persisted
```

class UserDictFactory(DictFactory) ...

```
Generate _inputs_ for an API or a function

<aside class="notes">
</aside>
```

Execution or Act

- Control Time
- Test doubles: Mocks and other
- Interceptors

Freezegun

```python
from datetime import datetime
from freezegun import freeze_time

@freeze_time("2012-01-14")
def test_2012():
    assert datetime.now() == datetime(2012, 1, 14)

    with freeze_time("2018-01-01"):
        assert datetime.now() > datetime(2012, 1, 4)
```

```
Library: freezegun
```

Freezer

```python
from datetime import datetime

def test_freezer_move(freezer):
    now = datetime.now()
    freezer.move_to('2017-05-20')
    later = datetime.now()
    assert now != later

@pytest.mark.freeze_time('2017-05-21')
def test_freeze_decorator():
    assert datetime.now() == datetime(2017, 5, 21)
```

```
Fixture: freezer
```

Other date and time libraries

timeago - moment - pytime - arrow

```
timeago.format(timedelta(seconds = 60 * 3.4))  # 3 minutes ago
moment.date(2012, 12, 19).add(hours=1, minutes=2, seconds=3)  # n
pytime.next_month('2015-10-1')  # again, no-more-deltas
arrow.utcnow().span('hour')  # 2 datetimes in one line
```

Execution: test doubles

Doubles replace some element to provide the desired behaviour

Test doubles types

Mock library

In python most doubles are implemented by the mock library

```
Library: mock - Plugin: pytest-mock  - Fixture: mocker
```

Python mocks

```
def test_mock_patching(mocker):
    url = "https://2021.es.pycon.org/"
    mocked = mocker.patch.object(requests, "get", return_value="i
    mocker.patch.object(requests, "post", side_effect=ForbiddenEr
    response = requests.get(url)
    assert response == "intercepted"
    assert mocked.called_once()
```

Stubs

```
def test_stubbing(monkeypatch):

    def mock_exist(value):
        print(f"{value} exists")
        return True

    monkeypatch.setattr(os.path, 'exists', mock_exist)
    assert os.path.exists("/believe/me/I/exist")
```

Spies

```
def test_with_spy(mocker):
    url = "https://2021.es.pycon.org/"
    spy = mocker.spy(requests, "get")
    response = requests.get(url)
    assert response.status_code -= 200
    spy.assert_called_once(), spy.mock_calls
    spy.assert_called_with(url)
```

It does not intercept, only registers callbacks

Interceptors

```python
def test_with_http_interceptor(requests_mock):
    # arranges
    url = "http://tests.com"
    requests_mock.get(url, json={"key": "value"})
    # action
    response = requests.get(url)

    assert "key" in response.json()
```

Intercepts callbacks, returns what you need

```
Libray: requests_mock
```

Validation or assert

```
def test_one():
    expected = 5
    result = system_under_test()
    assert result
    assert result is not None
    assert result == expected
    assert result > 3
```

Assertion error messages

```
response = requests.get(url)
assert response.json() == expected, response.text()
```

In case of `AssertionError`

we expose some extra information

Comparisons

```
import pytest, math

assert 2.2 == pytest.approx(2.3, 0.1)
assert math.isclose(2.2, 2.20001, rel_tol=0.01)
```

Don’t lose time on unimportant differences

Comparisons: list and sets

```
my_list = [1, 1, 2, 3, 4]
other_list = [4, 3, 2, 1]

list_without_duplicates = list(set(my_list))
diff = set(my_list) ^ set(other_list)
assert not diff
```

Don’t lose your head around repeated values

nor with loops comparing list

Comparison: dicts

```
from deepdiff import DeepDiff

def test_dicts(parameter, expected):
    result = system_under_test(parameter)
    diff = DeepDiff(result, expected, exclude_paths=(f"root['upda
    assert not diff, diff
```

Do not lose time ordering responses

Ignore painful fields like dates

```
Library: deepdiff
```

Assert at the end

```
def test_delayed_response(requests_mock):
    url = "http://tests.com"
    requests_mock.get(url, json={"key": "value"})

    response = requests.get(url)

    expect(response.status_code == 200, response.status_code)
    expect(response.json() == {}, response.text)
    assert_expectations()
```

```
Library: delayed-assert
```

Delayed assert prompt

```
    def assert_expectations():
        'raise an assert if there are any failed expectations'
        if _failed_expectations:
>           assert False, _report_failures()
E           AssertionError:
E
E           assert_expectations() called at
E           "/home/hector/Code/personal/effective_testing/tests/u
E
E           Failed Expectations : 1
E
E           1: Failed at "/home/hector/Code/personal/effective_te
E               ErrorMessage:    {"key": "value"}
E               expect(response.status_code == 200, response.stat

../../../.local/share/virtualenvs/effective_testing-OL_w0ui8/lib/
```

Log validation

Log are key for several reason

- Monitoring and metrics
- Treaceability and error control
- Especially important on microservices and serverless

Log capture

```
def test_log_capture(request, caplog):
    logger = logging.getLogger(request.node.name)
    caplog.set_level("INFO")
    dtt = "2021-10-02 10:55:00"
    msg = "captured message"
    expected_message = f"{dtt} INFO module:{request.node.name} {m
    with freeze_time(dtt):
   logger.info("captured message")

    with caplog.disabled():
    logger.info ("Any log here will not be captured")
    some_function_with_logs()
    breakpoint()

    with freeze_time(dtt):
   logger.info("captured message")
```

```
Fixture: caplog, capsys, capfd
```

Extra ball: chasing errors

- Tests are little environments you can reuse to look for unexpected errors
- Debugging in live servers is not always possible nor easy

```
pip install ipdb
PYTHONBREAKPOINT=ipdb.set_trace pytest -m current -s
```

```
Use your IDE built-in breakpoints and debugger
```

Library: ipdb, IDEs

Anti-patterns

- Too many mocks
- Fighting the framework
- Sequential tests
- Dirty sources
- Test Python or a library

https://www.softwaretestingmagazine.com/knowledg
patterns-in-software-testing/

https://dzone.com/articles/unit-testing-anti-
patterns-full-list

Final ideas

- “Tests are a waste of time”
  - Actually they save you a lot of time
- Treat tests as first-class citizens:
  - style, docstrings, comments ...
- There is no small project that is worth having tests
- Think on your peers and your future self
- Remember the test pyramdg, don’t put all your stakes in one type of tests

Recommendations

- If something is taking too long to do
  - there’s a library or a recipe that can help you
- Read and revisit the documentation
  - you will always find something new
- Frameworks help you, check their test documentation

Summary

Fixtures: monkeypatch, mocker, requests_mock, caplog, parametrize, mark *Libraries*: factoryboy, faker, deepdict, freezegun, moment, ipdb

https://docs.pytest.org/

https://docs.pytest.org/en/latest/reference/plugin_lis

Thanks!

Thanks for your attention, hope you like it

Any question, suggestion ... ?

https://github.com/hectorcanto/pytest-samples

https://www.slideshare.net/HectorCant